



UNIVERSIDAD  
AUTÓNOMA  
METROPOLITANA  
Unidad Cuajimalpa

DCNI  
División de Ciencias  
Naturales e Ingeniería

# Desarrollo de software a gran escala

**Autores:**

Dr. Pedro Pablo González Pérez  
Dra. María del Carmen Gómez Fuentes  
Dr. Jorge Cervantes Ojeda



Docencia

# **Desarrollo de Software a Gran Escala**



Casa abierta al tiempo

**UNIVERSIDAD AUTÓNOMA METROPOLITANA**  
Unidad Cuajimalpa

**Autores**

Pedro Pablo González Pérez  
María del Carmen Gómez Fuentes  
Jorge Cervantes Ojeda

**Departamento de Matemáticas Aplicadas y Sistemas  
División de Ciencias Naturales e Ingeniería  
Universidad Autónoma Metropolitana, Unidad Cuajimalpa**

**Editada por:**

**UNIVERSIDAD AUTONOMA METROPOLITANA**

Prolongación Canal de Miramontes 3855,  
Quinto Piso, Col. Ex Hacienda de San Juan de Dios,  
Del. Tlalpan, C.P. 14787, México D.F.

**Desarrollo de Software a Gran Escala**

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión en ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares.

**ISBN: 978-607-28-2846-9**

**Primera edición 2023**

# Contenido

<b>CONTENIDO</b> .....	<b>I</b>
<b>INTRODUCCIÓN</b> .....	<b>V</b>
<b>RELACIÓN DEL CONTENIDO CON EL PROGRAMA DE ESTUDIO DE LA UEA “DESARROLLO DE SOFTWARE A GRAN ESCALA”</b> .....	<b>IX</b>
<b>OBJETIVOS</b> .....	<b>XI</b>
OBJETIVO GENERAL:.....	XI
OBJETIVOS ESPECÍFICOS: .....	XI
CONOCIMIENTOS PREVIOS:.....	XI
<b>CAPÍTULO I     LOS SISTEMAS DE SOFTWARE A GRAN ESCALA</b> .....	<b>12</b>
I.1   INTRODUCCIÓN .....	12
I.2   IMPACTO DE LAS CARACTERÍSTICAS DEL SOFTWARE A GRAN ESCALA EN EL PROCESO DE DESARROLLO.....	13
I.3   INFRAESTRUCTURA DE SOPORTE PARA EL DESARROLLO DE SOFTWARE A GRAN ESCALA.....	14
I.4   ASPECTOS A CONSIDERAR EN EL DESARROLLO DE SOFTWARE A GRAN ESCALA.....	15
I.5   CONOCIMIENTOS Y HABILIDADES PREVIOS REQUERIDOS PARA INICIAR CON EL DESARROLLO DE SOFTWARE A GRAN ESCALA	15
I.6   CASO DE ESTUDIO .....	16
I.7   REFERENCIAS DEL CAPÍTULO.....	20
<b>CAPÍTULO II     ROLES Y RESPONSABILIDADES EN UN EQUIPO DE DESARROLLO DE SOFTWARE A GRAN ESCALA</b>	<b>21</b>
II.1   ROLES Y RESPONSABILIDADES .....	21
II.2   EL RESPONSABLE DEL PROYECTO .....	22
II.3   LOS LÍDERES DE EQUIPO .....	22
II.4   LOS ANALISTAS DE SISTEMAS.....	23
II.5   LOS ARQUITECTOS DE SOFTWARE.....	23
II.6   LOS ENCARGADOS DEL DISEÑO DETALLADO .....	23
II.7   LOS PROGRAMADORES.....	24
II.8   LOS RESPONSABLES DE LAS BASES DE DATOS .....	24
II.9   LOS RESPONSABLES DE LA VERIFICACIÓN Y PRUEBAS .....	25
II.10   LOS RESPONSABLES DE SOPORTE TÉCNICO .....	25
II.11   CONFORMACIÓN DE LOS EQUIPOS DE DESARROLLO EN PROYECTOS DE SOFTWARE A GRAN ESCALA.....	25
II.12   CASO DE ESTUDIO .....	27
II.13   REFERENCIAS DEL CAPÍTULO .....	29

<b>CAPÍTULO III</b>	<b>EL PROCESO DE REQUERIMIENTOS</b> .....	<b>30</b>
III.1	EL ALCANCE DEL ANÁLISIS DE REQUERIMIENTOS.....	30
III.2	ACTIVIDADES DEL PROCESO DE REQUERIMIENTOS.....	31
III.3	CASOS DE ESTUDIO.....	42
III.4	REFERENCIAS DEL CAPÍTULO.....	58
<b>CAPÍTULO IV</b>	<b>GESTIÓN DE LOS REQUERIMIENTOS NO FUNCIONALES</b> .....	<b>59</b>
IV.1	LOS REQUERIMIENTOS NO FUNCIONALES.....	59
IV.2	CASO DE ESTUDIO.....	61
IV.3	REFERENCIAS DEL CAPÍTULO.....	65
<b>CAPÍTULO V</b>	<b>DISEÑO ARQUITECTÓNICO Y DISEÑO DE LA INTERACCIÓN ENTRE INTERFACES GRÁFICAS DE USUARIO</b> .....	<b>66</b>
V.1	ARQUITECTURA LÓGICA DEL SISTEMA DE SOFTWARE.....	66
V.2	MODELO ARQUITECTÓNICO DE SISTEMAS INTERACTIVOS.....	68
V.3	MODELO ARQUITECTÓNICO DE PIZARRA ( <i>BLACKBOARD ARCHITECTURE</i> ).....	78
V.4	LA ARQUITECTURA FÍSICA DEL SOFTWARE.....	85
V.5	VISTAS DE LA ARQUITECTURA DEL SOFTWARE.....	90
V.6	INTERACCIONES ENTRE LAS INTERFACES DE USUARIO CON DIAGRAMAS DE TRANSICIÓN ENTRE INTERFACES DE USUARIO.....	94
V.7	DIAGRAMA DE SECUENCIAS DETALLADO EN EL DISEÑO DETALLADO.....	96
V.8	CASO DE ESTUDIO 1.....	97
V.9	CASOS DE ESTUDIO 2.....	107
V.10	REFERENCIAS DEL CAPÍTULO.....	119
<b>CAPÍTULO VI</b>	<b>EXTENSIONES Y VARIANTES DEL MODELO ARQUITECTÓNICO DE SISTEMAS DE SOFTWARE INTERACTIVOS</b> .....	<b>121</b>
VI.1	INTRODUCCIÓN.....	121
VI.2	ARQUITECTURA MODELO-VISTA-PRESENTADOR (MODEL-VIEW-PRESENTER).....	122
VI.3	ARQUITECTURA PRESENTACIÓN-ABSTRACCIÓN-CONTROL (PRESENTATION-ABSTRACTION-CONTROL).....	126
VI.4	ARQUITECTURA MODELO-VISTA-CONTROLADOR JERÁRQUICO (HIERARCHICAL MODEL-VIEW-CONTROLLER).....	128
VI.5	ARQUITECTURA DE CUATRO CAPAS MODELO-VISTA-CONTROLADOR-SERVICIOS.....	129
VI.6	REFERENCIAS DEL CAPÍTULO.....	131
<b>CAPÍTULO VII</b>	<b>MODULARIZACIÓN Y DISEÑO DE COMPONENTES</b> .....	<b>132</b>
VII.1	MODULARIZACIÓN, COHESIÓN Y ACOPLAMIENTO.....	132
VII.2	DISEÑO DE COMPONENTES.....	136
VII.3	DIAGRAMAS DE ACTIVIDADES.....	153
VII.4	CASOS DE ESTUDIO.....	156
VII.5	REFERENCIAS DEL CAPÍTULO.....	177
<b>CAPÍTULO VIII</b>	<b>EL USO DE PATRONES EN LA INGENIERÍA DEL SOFTWARE: PATRONES DE DISEÑO</b>	<b>178</b>
VIII.1	EL USO DE PATRONES EN LA INGENIERÍA DEL SOFTWARE.....	178
VIII.2	TAXONOMÍAS DE PATRONES.....	181
VIII.3	PATRONES DE DISEÑO.....	183
VIII.4	CASOS DE ESTUDIO.....	212
VIII.5	REFERENCIAS DEL CAPÍTULO.....	221
<b>CAPÍTULO IX</b>	<b>GESTIÓN DE LA CONFIGURACIÓN</b> .....	<b>222</b>
IX.1	IDENTIFICACIÓN DE LOS ELEMENTOS DE LA CONFIGURACIÓN DEL SISTEMA.....	222
IX.2	GESTIÓN DE CAMBIOS.....	223
IX.3	GESTIÓN DE VERSIONES.....	225

IX.4	CASO DE ESTUDIO .....	228
IX.5	REFERENCIAS DEL CAPÍTULO .....	230
<b>CAPÍTULO X</b>	<b>GESTIÓN DE LA CALIDAD Y LAS PRUEBAS .....</b>	<b>231</b>
X.1	INTRODUCCIÓN .....	231
X.2	PRUEBAS DEL SOFTWARE .....	233
X.3	GESTIÓN DE LAS PRUEBAS .....	234
X.4	CASO DE ESTUDIO .....	238
X.5	REFERENCIAS DEL CAPÍTULO .....	244
<b>CAPÍTULO XI</b>	<b>MÉTRICAS DEL PROCESO DE DESARROLLO DE SOFTWARE Y DEL SOFTWARE .....</b>	<b>245</b>
XI.1	PRINCIPIOS DE MEDICIÓN .....	245
XI.2	MÉTRICAS DEL PROCESO DE DESARROLLO Y MÉTRICAS DEL SOFTWARE .....	246
XI.3	ESTIMACIÓN DE LOS PUNTOS DE FUNCIÓN .....	248
XI.4	ESTIMACIÓN DEL TAMAÑO DEL SOFTWARE EN LÍNEAS DE CÓDIGO A PARTIR DE LOS PUNTOS DE FUNCIÓN .....	252
XI.5	ESTIMACIÓN DEL ESFUERZO PERSONAS-MES Y DE LA PLANEACIÓN EN MESES .....	253
XI.6	MÉTRICAS ESTÁTICAS Y MÉTRICAS DINÁMICAS .....	256
XI.7	CASO DE ESTUDIO .....	257
XI.8	REFERENCIAS DEL CAPÍTULO .....	263
<b>CAPÍTULO XII</b>	<b>GESTIÓN DEL MANTENIMIENTO .....</b>	<b>265</b>
XII.1	TIPOS DE MANTENIMIENTO .....	265
XII.2	GESTIÓN DEL MANTENIMIENTO .....	266
XII.3	GESTIÓN DE LOS CAMBIOS Y DE LOS REPORTES DE ERROR DURANTE EL MANTENIMIENTO .....	268
XII.4	REFERENCIAS DEL CAPÍTULO .....	270
<b>APÉNDICE .....</b>	<b>271</b>	
A.	GLOSARIO DE ABREVIACIONES .....	271
B.	GLOSARIO DE TÉRMINOS .....	272



# Introducción

El desarrollo de software a gran escala es un proceso caracterizado por un mayor nivel de dificultad en comparación con el desarrollo de software a mediana o a pequeña escala. Cuando hablamos de software a gran escala nos referimos a aquel tipo de software que exhibe gran parte de los siguientes aspectos, muchos de los cuales necesariamente tendrán que ser considerados y resueltos durante el proceso de desarrollo:

- 1) Funcionalidad extensa y compleja
- 2) Dependencia de una gran variedad de requerimientos no funcionales, comúnmente conocidos como “restricciones”
- 3) Necesidad de procesar una gran cantidad de información (almacenamiento, recuperación, consulta, inferencias lógicas, etc.)
- 4) Necesidad de alta modularización y desacoplamiento entre vistas-interacción, lógica y modelos de datos
- 5) El procesamiento, la información y el control se encuentran distribuidos
- 6) Integración de diferentes sistemas de software, hardware y comunicaciones.

Los aspectos antes relacionados impactan significativamente en el nivel de dificultad del proceso de desarrollo de software, al requerirse de un numeroso equipo de desarrollo, diferentes roles dentro del equipo, el conocimiento y experiencia especializados en las diferentes áreas de ingeniería de software y, comúnmente, de un prolongado período de desarrollo.

Para comprender cómo impactan algunos de estos aspectos en el proceso de desarrollo de software, veamos los siguientes ejemplos:

- a) **Funcionalidad extensa y compleja.** Es necesario invertir una importante cantidad de tiempo en el proceso de requerimientos y su gestión de riesgos. De igual manera las fases de diseño, la codificación y las pruebas también requerirán de un mayor esfuerzo.
- b) **Dependencia de una gran variedad de requerimientos no funcionales.** Implica que, durante el proceso de desarrollo, además del análisis, diseño y codificación de los requerimientos funcionales, también será necesario contender con requerimientos no funcionales tales como: confiabilidad, robustez, seguridad, desempeño, portabilidad, facilidad de mantenimiento, entre otros.
- c) **Necesidad de procesar una gran cantidad de información.** Cuando se trabaja con grandes volúmenes de información, se suele recurrir a técnicas avanzadas de captura,

validación, almacenamiento, consulta y recuperación, tales como, la minería de datos y el análisis inteligente de la información. El uso de estas técnicas implica el diseño y ejecución de casos de prueba adicionales.

Tomando en consideración lo antes expuesto, el presente material pretende contribuir en la identificación temprana de los problemas que se presentan durante el desarrollo de productos de software a gran escala; la comprensión de los roles y responsabilidades de las personas que intervienen en el desarrollo de un proyecto de software a gran escala; la comprensión y uso de las métricas que ayuden a la estimación y planeación en el desarrollo de los sistemas de software a gran escala; el uso de modelos y técnicas que permitan una explícita y comprensible especificación de los requerimientos; la selección de la arquitectura adecuada sobre la cual se erigirá el sistema de software; la importancia y aplicación de las técnicas de modularización y desacoplamiento durante el diseño de los componentes del sistema de software; y la comprensión de las tareas de la gestión de la configuración, gestión de calidad y pruebas, así como del mantenimiento, en el desarrollo de sistemas de software a gran escala.

Para los fines de este material, nuestro interés se centrará en la construcción de software a gran escala, enfatizando las fases de estimación y planeación, especificación de requerimientos, diseño arquitectónico y patrones de arquitectura, diseño detallado y patrones de diseño e implementación. Así mismo, todos aquellos métodos, artefactos y patrones que faciliten y beneficien el desarrollo de estas etapas de la construcción de un sistema de software a gran escala serán ilustrados a través de dos casos de estudio demostrativos. El primer caso de estudio se refiere a un sistema computacional para el monitoreo y detección de fugas en tiempo real en ductos que transportan hidrocarburos. El segundo caso de estudio está relacionado con la plataforma bioinformática *Evolution*, un ambiente computacional para el modelado y simulación del plegamiento de proteínas. Ambos sistemas computacionales caen en la categoría de sistemas de software a gran escala, aunque los mismos no exhiban el mismo grado de complejidad.

El Capítulo I ofrece un panorama de los sistemas de software a gran escala y del proceso de desarrollo que conlleva la construcción de este tipo de software, enfatizando en aspectos tales como sus características y el impacto en el proceso de desarrollo, la infraestructura de soporte requerida, los aspectos claves a considerar en el proceso de desarrollo, así como los conocimientos y habilidades previos requeridos para iniciar con el desarrollo de software a gran escala. El capítulo finaliza con un caso de estudio, que ilustra la caracterización de un sistema de software como sistema de software a gran escala.

En el Capítulo II se discute la integración de los equipos de desarrollo de software a gran escala, enfatizando en los diferentes roles y responsabilidades requeridos. En este capítulo también se presentan las principales variantes para la conformación de los equipos de desarrollo; cuando el proyecto en cuestión exige contar con más de un equipo de desarrollo. El capítulo concluye con un caso de estudio dedicado a la conformación de los equipos de desarrollo en la construcción del Subsistema de Procesamiento Integral de Datos, del Sistema de Detección de Fugas y Tomas Clandestinas en Ductos que Transportan Hidrocarburos.

Las principales actividades, técnicas y modelos envueltos en el análisis de requerimientos son expuestos en el Capítulo III. El capítulo enfatiza de forma particular en los modelos y

técnicas utilizados durante la especificación de los requerimientos. El capítulo finaliza con dos casos de estudio que ilustran la aplicación de algunos de estos modelos y técnicas durante la especificación de los requerimientos en el Subsistema de Procesamiento Integral de Datos, del Sistema de Detección de Fugas y Tomas Clandestinas en Ductos que Transportan Hidrocarburos, y la plataforma bioinformática *Evolution*.

El Capítulo IV aborda la gestión de los requerimientos no funcionales o restricciones, un aspecto crucial que caracteriza los sistemas de software a gran escala, abarcando tanto a los requerimientos no funcionales que caracterizan la ejecución o funcionamiento global del sistema de software, como a los requerimientos no funcionales que caracterizan al sistema de software como producto. El capítulo concluye ilustrando parte de la gestión de los requerimientos no funcionales en los dos casos de estudio presentados en el Capítulo III.

El diseño arquitectónico, como fase clave del desarrollo de software a gran escala, es discutido en el Capítulo V, el cual inicia definiendo e ilustrando los conceptos de arquitectura lógica (arquitectura del software) y arquitectura física (componentes de hardware), indisolublemente ligados al diseño arquitectónico. El capítulo se centra en dos tipos de arquitectura lógica ampliamente utilizadas en el desarrollo de software a gran escala: el modelo arquitectónico de sistemas interactivos, comúnmente referido como modelo-vista-controlador (MVC) y el modelo arquitectónico de pizarra o repositorio (del inglés, *blackboard architecture*). Por otra parte, se discuten e ilustran los tres tipos clave de arquitectura física: sistemas monolíticos, sistemas distribuidos y sistemas cliente-servidor web. El capítulo finaliza mostrando parte del diseño arquitectónico en los dos casos de estudio presentados en el Capítulo III.

El Capítulo VI retoma el modelo arquitectónico de sistemas interactivos (modelo-vista-controlador) y discute e ilustra algunas variantes y extensiones del mismo, tales como: modelo-vista-presentador (del inglés, *model-view-presenter*), presentación-abstracción-control (del inglés, *presentation-abstraction-control* (PAC)), modelo-vista-controlador jerárquico (del inglés, *hierarchical model-view-controller*) y la arquitectura de cuatro capas modelo-vista-controlador-servicios. El capítulo finaliza con una propuesta de arquitectura alternativa (la arquitectura PAC), la cual también fue considerada durante el diseño arquitectónico del Subsistema de Procesamiento Integral de Datos, del Sistema de Detección de Fugas y Tomas Clandestinas en Ductos que Transportan Hidrocarburos.

La modularización y diseño detallado son discutidos en el Capítulo VII. El capítulo se centra en el papel de la modularización, cohesión y acoplamiento en el diseño de software, ilustrando estas técnicas y principios a partir de varios escenarios. El capítulo finaliza mostrando elementos del diseño detallado en los dos casos de estudio, ya referidos, en el Capítulo III.

El Capítulo VIII está dedicado a una importante técnica (también considerada una buena práctica) en el desarrollo de software a gran escala, esta es, el uso de patrones. El capítulo se centra en la utilidad de los patrones de diseño, describiendo y ejemplificando dos categorías de este tipo: los patrones de creación y los patrones estructurales. El capítulo concluye ilustrando el uso de los patrones de diseño en los dos casos de estudio presentados en el Capítulo III.

La gestión de la configuración, una actividad crucial a ser considerada durante el proceso de desarrollo de software a gran escala, es tratada en el Capítulo IX. El capítulo inicialmente se centra en la identificación de los elementos de la configuración del sistema, pasando posteriormente a tratar la gestión de cambios y la gestión de versiones. El capítulo finaliza ilustrando elementos de la gestión de versiones en los dos casos de estudio presentados en el Capítulo III.

El Capítulo X proporciona una introducción a la gestión de la calidad y de las pruebas en el desarrollo de software a gran escala. El capítulo finaliza mostrando parte de la metodología de la gestión de pruebas del Subsistema de Procesamiento Integral de Datos, del Sistema de Detección de Fugas y Tomas Clandestinas en Ductos que Transportan Hidrocarburos.

En el Capítulo XI está dedicado a los principios de medición y métricas, y su importancia en la estimación del tamaño del software, así como en la estimación y planeación del proceso de desarrollo de software. Entre otras técnicas de estimación, el capítulo se refiere al método IBM para la estimación de los puntos de función, a la estimación del tamaño en líneas de código a partir del tamaño en puntos de función, al método de estimación de primer orden de Jones, y al modelo COCOMO (del inglés, *Constructive Cost Model*). El capítulo finaliza describiendo el proceso de estimación llevado a cabo durante el desarrollo de la plataforma bioinformática *Evolution*.

Finalmente, el Capítulo XII proporciona una introducción a la gestión del mantenimiento del software, haciendo referencia a los principales tipos de mantenimiento, así como a las actividades involucradas en la gestión del mantenimiento. El capítulo finaliza haciendo referencia de los aspectos de la gestión del mantenimiento de la plataforma bioinformática *Evolution*.

# Relación del contenido con el programa de estudio de la UEA “Desarrollo de Software a Gran Escala”

El contenido de este libro se basa íntegramente en el programa de estudio de la Unidad de Enseñanza Aprendizaje (UEA) Desarrollo de Software a Gran Escala, de la Licenciatura de Ingeniería en Computación, el cual se relaciona a continuación, integrando, además, temas claves que se ofrecen en las UEA Análisis y Diseño Orientado a Objetos, y Proyecto de Ingeniería de Software II, ambas pertenecientes al Plan de Estudios antes mencionado.

## CONTENIDO SINTÉTICO DE LA UEA DESARROLLO DE SOFTWARE A GRAN ESCALA

1. Los problemas que se presentan durante el desarrollo de proyectos de software a gran escala.
  - 1.1. Situación actual sobre los resultados de entrega de proyectos de software.
  - 1.2. Problemas de las metodologías tradicionales.
  - 1.3. Tipos de errores y riesgos más comunes en el desarrollo del software.
2. Roles y responsabilidades en el desarrollo de proyectos de software a gran escala.
  - 2.1. El responsable del proyecto y los líderes de equipo.
  - 2.2. Los analistas del sistema.
  - 2.3. Los arquitectos del software y los encargados del diseño detallado.
  - 2.4. Los programadores y los responsables de la base de datos.
  - 2.5. Probadores y soporte técnico.
3. Diseño arquitectónico.
  - 3.1. Conceptos de la arquitectura del software.
  - 3.2. Patrones arquitectónicos.
  - 3.3. Modelado arquitectónico.
4. Diseño de componentes.
  - 4.1. Modularización.
  - 4.2. Comunicación y dependencia entre módulos (ensamblaje).
  - 4.3. Modelado del diseño.
5. La gestión de la configuración.
  - 5.1. Identificación de los elementos de la configuración del sistema.
  - 5.2. Control de versiones.
  - 5.3. Control de cambios.

6. Introducción a la gestión de calidad y pruebas.
  - 6.1. Introducción a la gestión de calidad en los proyectos a gran escala.
  - 6.2. Introducción a la gestión de pruebas en los proyectos a gran escala.
7. Métricas.
  - 7.1. Principios de medición.
  - 7.2. Métricas estáticas y dinámicas.
  - 7.3. Tipos de métricas de software.
  - 7.4. Herramientas para métricas de software.
8. Las actividades de mantenimiento y soporte técnico.
  - 8.1. Tipos de mantenimiento.
  - 8.2. Gestión de mantenimiento.
  - 8.3. Software de soporte técnico.

# Objetivos

## Objetivo general:

Conocer y aplicar las técnicas de la ingeniería de software para desarrollar con calidad un sistema de software a gran escala.

## Objetivos específicos:

1. Identificar los problemas que se presentan durante el desarrollo de productos de software a gran escala.
2. Comprender las responsabilidades de las diferentes personas que intervienen en el desarrollo de un proyecto de software a gran escala.
3. Identificar las arquitecturas más adecuadas para el desarrollo de software a gran escala.
4. Aplicar los patrones de arquitectura y de diseño en el desarrollo de sistemas de software a gran escala.
5. Aplicar el método de modularización y ensamblaje en el diseño de los componentes.
6. Comprender las tareas de la gestión de la configuración, gestión de calidad y pruebas, de mantenimiento y de soporte técnico.
7. Comprender las métricas que ayudan a determinar cuáles son los sistemas de software a gran escala.

## Conocimientos previos:

Para comprender y aprovechar mejor este material, se recomienda que el lector posea conocimientos y habilidades en el desarrollo de software orientado a objetos.

# Capítulo I Los Sistemas de Software a Gran Escala

## I.1 Introducción

Cuando hablamos de software a gran escala [1-5] nos referimos a aquel tipo de software que exhibe gran parte de los siguientes aspectos, los cuales necesariamente tendrán que ser considerados y resueltos durante el proceso de desarrollo del mismo:

- Funcionalidad extensa y compleja que requiere ser implementada en diferentes componentes de forma simultánea por equipos de desarrollo independientes.
- Dependencia de una gran variedad de requerimientos no funcionales, comúnmente conocidos como restricciones.
- Alta capacidad de procesamiento de grandes cantidades de información en poco tiempo (tráfico): almacenamiento, recuperación, consulta, inferencias, análisis inteligente de datos, etc.
- La complejidad de las tareas que deben ser implementadas conlleva a la integración y uso de diferentes técnicas, modelos y métodos provenientes de áreas tales como las matemáticas, la física, la biología, la ingeniería y la inteligencia artificial, entre otras.
- Comúnmente, el procesamiento, la información y el control se encuentran distribuidos.
- El sistema resultante es la integración de diferentes componentes de software, hardware y comunicaciones.

### I.1.1 Impacto de las características del software a gran escala en el proceso de desarrollo

El desarrollo de software a gran escala requiere de un gran equipo de desarrollo, de diferentes roles dentro del equipo, del conocimiento y experiencia en la ingeniería de software, en las técnicas, modelos y métodos en los que se debe basar el sistema, así como en los dominios de tareas que debe ejecutar el mismo, en una rigurosa planeación, y en un prolongado período de desarrollo [1-5]. En la Tabla 1.1 se relaciona el impacto de las características de este tipo de software en el proceso de desarrollo.

**Tabla 1.1.** Impacto de las características del software a gran escala en el proceso de desarrollo

Característica del software a gran escala	Impacto en el proceso de desarrollo
Funcionalidad extensa y compleja (equipos de desarrollo independientes)	<ul style="list-style-type: none"> <li>• Dificultad en la recolección, definición y especificación de los requerimientos.</li> <li>• Necesidad de identificar, evaluar y mitigar los riesgos atribuibles a la especificación de los requerimientos.</li> <li>• Generación de una gran cantidad de código.</li> </ul>
Dependencia de una gran variedad de requerimientos no funcionales	<ul style="list-style-type: none"> <li>• Necesidad de diseño de pruebas de confiabilidad, robustez, seguridad, desempeño, recuperación, portabilidad, entre otros.</li> </ul>
Alta capacidad de procesamiento de grandes cantidades de información en poco tiempo	<p>Necesidad de diseño de pruebas de:</p> <ul style="list-style-type: none"> <li>• Velocidad de acceso a datos.</li> <li>• Capacidad de procesamiento.</li> <li>• Capacidad de almacenamiento.</li> </ul> <p>Considerar el uso de técnicas de minería de datos y análisis inteligente de la información.</p>
Distribución del procesamiento y de la información	<p>Diseño de arquitecturas y tipos de procesamiento complejos, tales como:</p> <ul style="list-style-type: none"> <li>• Cliente-Servidor.</li> <li>• Cliente-Servidor multicapa.</li> <li>• <i>Blackboard Architecture</i>.</li> <li>• Sistemas cooperativos distribuidos.</li> <li>• Procesamiento paralelo.</li> </ul>
Integración de diferentes sistemas de software, hardware y comunicaciones	<ul style="list-style-type: none"> <li>• Interacción con sistemas remotos (por ejemplo: sistema de autorización de pago, bases de datos (BD) en servidores remotos, sistema de control de inventario localizado en un servidor remoto, etc.).</li> <li>• Requerimientos de hardware periférico (por ejemplo: lectores ópticos, sensores, scanner, actuadores, sistemas de control de acceso, etc.).</li> <li>• Diferentes canales de comunicación (por ejemplo: redes alámbricas, redes inalámbricas, infrarrojo, <i>bluetooth</i>, etc.).</li> </ul>
Gestión del personal	<ul style="list-style-type: none"> <li>• Diferentes roles en el equipo de desarrollo.</li> <li>• Interacción, comunicación y colaboración en el equipo de desarrollo.</li> </ul>

- |  |   |
|--|---|
|  | <ul style="list-style-type: none"> <li>• Actitudes personales y de grupo.</li> <li>• Importancia del conocimiento y de la experiencia en ingeniería de software.</li> <li>• Asociaciones y corresponsabilidades entre los profesionales.</li> </ul> |
|--|---|

## 1.2 Infraestructura de soporte para el desarrollo de software a gran escala

Existen diversos ambientes de desarrollo integrado, IDE (por sus siglas en inglés: Integrated Development Environment IDE) que manejan uno o varios lenguajes de programación. Hay de paga y de distribución libre. Los IDE se caracterizan principalmente por tener un editor de texto que facilite la codificación, un compilador con ayuda para la corrección de errores, y un depurador de código (*debugger*) que permite seguir la ejecución del programa para facilitar la detección de errores. Además, permiten el uso de bibliotecas, conexión a base de datos y la configuración modular del sistema. Con un buen IDE se pueden utilizar marcos de referencia (*frameworks*) que faciliten el desarrollo de las aplicaciones.

Los IDE estándares no manejan muchos de los problemas que surgen durante el desarrollo de software a gran escala. Este hecho conlleva a que la funcionalidad de los IDEs deba ser escalada (incrementada) con herramientas que permitan contender con tales problemas.

El desarrollo de software a gran escala requiere de la disponibilidad de tecnologías y herramientas escalables que soporten dicho desarrollo. Entre estas tecnologías y herramientas se encuentran aquellas orientadas a proporcionar soporte en:

- El aseguramiento de la calidad.
- La verificación y las pruebas.
- El mantenimiento.
- La liberación.
- Los requerimientos de soporte del cliente.

Algunos de los principales problemas que surgen durante el desarrollo de software a gran escala están estrechamente relacionados con la garantía de los siguientes requerimientos no funcionales:

- Desarrollo de software que soporte la reusabilidad y la extensibilidad.
- Organización y mantenimiento de los componentes de software.
- Administración de la memoria.
- Ejecución.
- Concurrencia/paralelismo.
- Restricciones de tiempo.
- Seguridad.
- Calidad.

### I.3 Aspectos a considerar en el desarrollo de software a gran escala

La Figura 1.1 ilustra los principales aspectos (actividades, métodos) a considerar en el desarrollo de un sistema de software a gran escala [2, 3, 5], los cuales serán abordados detalladamente a través de los diferentes capítulos que integran este material.



**Figura 1.1.** Aspectos a considerar en el desarrollo de un sistema de software a gran escala.

### I.4 Conocimientos y habilidades previos requeridos para iniciar con el desarrollo de software a gran escala

Aunque el desarrollo de sistemas de software a gran escala no está restringido al uso de la programación orientada a objetos – y por lo tanto, a las metodologías y tecnologías orientadas a objetos – resulta imposible no reconocer que son precisamente las metodologías y tecnologías orientadas a objetos las que proporcionan un soporte robusto para alcanzar requerimientos tales como arquitectura cliente-servidor web multicapa, máximo desacoplamiento entre los componentes o módulos que conforman el sistema, reusabilidad y extensibilidad; siendo éstos precisamente el común denominador de los requerimientos de sistemas de software a gran escala de hoy en día [1, 2].

Tomando estas ideas como un precedente verificado, el desarrollo de un sistema de software a gran escala requiere fundamentalmente de los siguientes conocimientos y habilidades:

- El dominio de una metodología de desarrollo de software caracterizada por un desarrollo iterativo, un desarrollo dirigido por los riesgos, un desarrollo incremental del sistema, y el uso de las tecnologías de objetos, las cuales son: análisis orientado a objetos (AOO), diseño orientado a objetos (DOO) y programación orientada a objetos (POO).
- Habilidades en el uso de *frameworks*, herramientas y modelos de análisis y diseño orientado a objetos (OO).
- Habilidades en el desarrollo de software en un lenguaje de programación robusto, que garantice tener el menor número de dependencias de implementación, concurrente y orientado a objetos.

## 1.5 Caso de estudio

### 1.5.1 Sistema de detección de fugas en ductos que transportan hidrocarburos. caracterización como sistema de software a gran escala.

Para ilustrar la complejidad del desarrollo de software a gran escala durante todo este material, se hará referencia de forma muy detallada al Subsistema de Procesamiento Integral de Datos (SPI), del Sistema de Detección de Fugas en Ductos que Transportan Hidrocarburos [6, 7]. Por supuesto, también abordaremos otros ejemplos de desarrollo de software a gran escala, como es el caso de la Plataforma Bioinformática Evolution [8], una herramienta computacional dedicada al modelado y simulación del plegamiento de secuencias de aminoácidos.

El Subsistema SPI tiene como principal objetivo aumentar la efectividad en la detección oportuna de fugas y tomas clandestinas en ductos que transportan hidrocarburos. Para lograr lo anterior, el subsistema SPI incorpora en su procesamiento integral dos importantes estrategias:

- i. Integración de las salidas producidas por cuatro subsistemas para la detección de fugas y tomas clandestinas: dos subsistemas basados en el cálculo de balance de masa, un subsistema basado en la tecnología fibra óptica, y otro subsistema basado en la tecnología acústica.
- ii. Integración de un grupo de valiosas técnicas de inteligencia artificial, tales como sistemas expertos, redes neuronales artificiales y mapas cognitivos difusos, para el análisis y clasificación de los eventos causados por fugas y tomas clandestinas. Tales eventos corresponden a las salidas de los cuatro sistemas previamente mencionados, las cuales integradas constituyen el conjunto de datos de entrada al subsistema SPI.

La Tabla 1.2 resume los principales rasgos que caracterizan al subsistema SPI como sistema de software a gran escala.

**Tabla 1.2.** Impacto de las características del software a gran escala en el proceso de desarrollo

<b>Característica del software a gran escala</b>	<b>Descripción</b>
Funcionalidad extensa y compleja	<p>a) Comunicación síncrona con los subsistemas remotos de detección de fugas, basados en balance de masa, fibra óptica y acústica, y con el sistema de control y monitoreo a nivel superior.</p> <p>b) Conformación y lectura del patrón integral de dato, a partir de los eventos reportados por los subsistemas de balance de masa, fibra óptica y acústica.</p> <p>c) Diagnóstico de fugas y tomas clandestinas, basada en técnicas de sistemas expertos.</p> <p>d) Diagnóstico de fugas y tomas clandestinas, basada en técnicas de redes neuronales artificiales multiestrato.</p> <p>e) Diagnóstico de fugas y tomas clandestinas, basada en redes neuronales artificiales multiestrato recurrentes.</p> <p>f) Diagnóstico de fugas y tomas clandestinas, basada en lógica difusa.</p> <p>g) Diagnóstico integral de fugas y tomas clandestinas, basado en los cuatro diagnósticos particulares antes relacionados.</p> <p>h) Optimización de los métodos de diagnóstico.</p> <p>i) Comunicación síncrona de los resultados del diagnóstico integral de fugas y tomas clandestinas al sistema de control y monitoreo a nivel superior.</p>
Dependencia de una gran variedad de requerimientos no funcionales	<p>a) Nivel de acceso. En función del tipo de usuario que tendrá acceso al subsistema SPI, se podrá detallar a qué módulos podrá acceder o qué acciones podría en un determinado momento efectuar. Para el subsistema SPI, deben definirse los niveles de acceso que correspondan a los siguientes usuarios o actores:</p> <ul style="list-style-type: none"> <li>▪ Operador del sistema de control y monitoreo a nivel superior– usuario pasivo, sólo recibe información visual.</li> <li>▪ Administrador - puede corresponder a dos modalidades: <ul style="list-style-type: none"> <li>i) para atender las cuestiones relacionadas con el mantenimiento del sistema, y ii) para ejecutar cada vez que sea necesario los algoritmos de optimización de los métodos de diagnóstico.</li> </ul> </li> </ul> <p>b) Interfaces con otros sistemas de software.</p> <ul style="list-style-type: none"> <li>▪ Interfaz con el sistema de control y monitoreo a nivel superior. Es importante determinar de forma precisa la forma en que dicho sistema podría proporcionar información sobre los ductos, básicamente cómo podría</li> </ul>

	<p>establecerse la comunicación con sus datos; por ejemplo, para obtener temperatura o presión en un determinado momento. Por otra parte, el subsistema SPI utilizará el subsistema de visualización del sistema de control y monitoreo a nivel superior para visualizar sus salidas.</p> <ul style="list-style-type: none"> <li>▪ Interfaz con el usuario. Se han identificado dos tipos principales de usuario: operador del sistema de control y monitoreo a nivel superior, y administrador del subsistema SPI. En base a lo anterior, será necesario definir las características de las interfaces para ambos tipos de usuario.</li> <li>▪ Interfaz con otros subsistemas. Cada uno de los subsistemas basados en balance de masa, fibra óptica y acústica que proporciona sus salidas al subsistema SPI, debe entregar los datos en un determinado formato, de tal manera que el sistema integral pueda procesar dicha información. En este caso, no solo nos referimos a la forma en que debe proceder la comunicación, sino también al formato que deberán tener los datos.</li> </ul> <p>c) Seguridad. La ejecución del subsistema SPI se realiza de forma independiente a la del sistema de control y monitoreo a nivel superior, con la finalidad de no entorpecer su operación. Los datos que el subsistema SPI manipule estarán bajo el control del administrador, y los resultados que el subsistema SPI arroje sólo podrán ser vistos por la persona que el administrador designe, que en primera instancia sería el operador del sistema de control y monitoreo a nivel superior.</p> <p>d) Robustez. En una primera etapa del desarrollo del subsistema SPI (a nivel prototipo), en función de la localización de los datos, el procesamiento integral se realiza de manera secuencial, para más adelante considerar la ejecución paralela según la localización de los datos que reciba el subsistema SPI.</p> <p>e) Confiabilidad. En función del número de subsistemas de entrada y de la confiabilidad de los datos de salida que éstos proporcionan, estará determinada la confiabilidad del subsistema SPI. En cierto sentido, la confiabilidad de las salidas del subsistema SPI dependerá de la confiabilidad de sus entradas y de la precisión del diagnóstico integral ejecutado.</p> <p>f) Sincronización. La sincronización de datos constituye una de las restricciones más fuertes a considerar durante el</p>
--	---

	<p>desarrollo del subsistema SPI. Será necesario considerar dos tipos de sincronización:</p> <ul style="list-style-type: none"> <li>- Las entradas al subsistema SPI provenientes de los subsistemas basados en balance de masa, fibra óptica y acústica.</li> <li>- La salida producida por el subsistema SPI dirigida al sistema de control y monitoreo a nivel superior.</li> </ul> <p>g) Contender con la omisión de datos de entrada. Cada uno de los sistemas de detección de fugas proporcionará sus datos de salida como entrada al subsistema SPI. Sin embargo, puede ocurrir que en un momento dado no todos los sistemas de detección de fugas se encuentren en operación, y aun así el subsistema SPI deberá efectuar el procesamiento de los datos de entrada y producir una salida al sistema de control y monitoreo a nivel superior, garantizando determinado nivel de confiabilidad.</p> <p>h) Restricciones de plataforma/sistema operativo.</p> <p>Debido a que la mayoría de los sistemas con los que interactuará el subsistema SPI son servidores basados en procesadores Intel, con sistema operativo (SO) Windows Server, entonces éste deberá ejecutarse en este tipo de plataforma. Las características de la plataforma también definen la velocidad de procesamiento y por lo tanto la velocidad con la que el subsistema SPI proporcionará sus salidas. Para no afectar el funcionamiento de estos sistemas, el subsistema SPI utilizará su propia plataforma. En una primera etapa (prototipo) se podrá usar una máquina virtual para hacer pruebas y determinar recursos necesarios.</p>
<p>Alta capacidad de procesamiento de grandes cantidades de información en poco tiempo</p>	<p>El procesamiento de información en el subsistema SPI involucra la recepción, pre-procesamiento, validación y almacenamiento de grandes volúmenes de información, recibidos en tiempo real de los subsistemas remotos de detección de fugas. Por otra parte, otras tareas que hacen que en SPI el procesamiento de información sea complejo son la recuperación de la información almacenada, la ejecución de los diagnósticos particulares e integral, y la conversión de datos, para su posterior envío al sistema de control y monitoreo a nivel superior.</p>
<p>Distribución del procesamiento de la información</p>	<p>En el subsistema SPI, tanto el procesamiento como el control se encontrarán distribuidos en diferentes nodos o componentes físicos. Es decir, la arquitectura física del subsistema SPI requerirá tres nodos físicos para el hospedaje y ejecución de los tres principales módulos que componen dicho subsistema: módulo de procesamiento integral, módulo de comunicaciones, y módulo de bases de datos.</p>

Integración de diferentes sistemas de software, hardware y comunicaciones	Una característica crucial que incrementa la complejidad del subsistema SPI es la diversidad de sistemas de software, hardware y comunicaciones que integra. Como ya mencionados, el subsistema SPI recibe entradas de datos en tiempo real, provenientes de cuatro subsistemas remotos de detección de fugas. Por otra parte, el propio subsistema SPI es un sistema distribuido cuyos componentes lógicos se ejecutan en tres nodos físicos que requieren de comunicación entre ellos. Finalmente, el subsistema SPI debe enviar los resultados de su procesamiento integral a otro sistema remoto, el sistema de control y monitoreo a nivel superior.
Gestión del personal	La gestión del personal es compleja, ya que va más allá de la gestión de los miembros del equipo de desarrollo; abarcando la comunicación y coordinación con los restantes equipos de desarrollo, uno dedicado a cada sistema remoto de detección de eventos relacionados con fugas y tomas clandestinas.

## 1.6 Referencias del capítulo

1. Cusumano, M.A. *The Factory Approach to Large-Scale Software Development: Implications for Strategy, Technology, and Structure*. Classic Reprints Series, 2017.
2. Garland, J., Anthony, R. *Large Scale Software Architecture. A Practical Guide Using UML*. Wiley, 2003.
3. Janakiram, D. *Building Large Scale Software Systems*. McGraw Hill Education, 2013.
4. Lakos, J. *Large-Scale C++ Software Design*. Addison Wesley, 1996.
5. Screerer, A. *Coordination in Large-Scale Agile Software Development: Integrating Conditions and Configurations in Multiteam Systems (Progress in IS)*. Springer, 2017.
6. González-Pérez, P.P., Sadovnychyy, A., Alarcón Ramos, L.A., González González, D., Elorza Gómez, K., Peña Falcón, J.L. Sistema para la detección de fugas y tomas clandestinas en ductos de transporte de gas y líquidos. Sistema Computacional de Procesamiento Integral de Datos (PI). Documento de Análisis y Diseño. Reporte Técnico. 2013.
7. González-Pérez, P.P., Schaum, A., Sadovnychyy, A., Bernal Jaquez, R., Méndez Gurrola, I.I., Alarcón Ramos, L.A., Rosales Cruz, L. Sistema para la detección de fugas y tomas clandestinas en ductos de transporte de gas y líquidos. Sistema Computacional de Procesamiento Integral de Datos (PI). Estado del Arte. Reporte Técnico. 2011.
8. Sánchez Gutiérrez, M.E. *Plataforma Bioinformática para el Estudio, Modelado y Simulación del Plegamiento de Proteínas*. Proyecto Terminal. Universidad Autónoma Metropolitana, Unidad Cuajimalpa. 2010.

# Capítulo II Roles y Responsabilidades en un Equipo de Desarrollo de Software a Gran Escala

## II.1 Roles y Responsabilidades

El desarrollo de software a gran escala requiere de equipos de desarrollo con una estructura óptima que garantice: diferentes roles y responsabilidades, interacción, comunicación y colaboración entre los miembros del equipo de desarrollo, actitudes personales y de grupo, disponibilidad de conocimiento y experiencia en las diferentes fases que abarca la ingeniería del software, y asociaciones y corresponsabilidades entre los profesionales [1-4].

Entre los principales roles y responsabilidades que deben caracterizar un equipo de desarrollo de software a gran escala, tal como se ilustra en la Figura 2.1, se encuentran [1-4]:

- Responsable del proyecto.
- Líderes de Equipo.
- Analistas de Sistema.
- Arquitectos del Software.
- Encargados del diseño detallado.
- Programadores.
- Responsables de las bases de datos.
- Responsables de las pruebas.
- Responsables del soporte técnico.



**Figura 2.1** Roles y responsabilidades en un equipo de desarrollo de software a gran escala.

## II.2 El responsable del proyecto

El responsable del proyecto funge como el máximo responsable de la definición del proyecto de desarrollo de software y de la asignación de todos los recursos (financieros, humanos, tecnológicos, etc.) para llevar a cabo el mismo, además de:

- Colaborar con el líder de proyecto en las tareas de administración, estimación y planeación del proyecto de desarrollo de software.
- Revisar y aprobar la planeación del proyecto de desarrollo de software.

## II.3 Los líderes de equipo

Como parte de su rol en el equipo de desarrollo, el líder de equipo debe ejercer las siguientes funciones:

- Como su nombre lo indica, funge como líder del equipo de desarrollo, el cual es integrado por analistas, arquitectos, encargados del diseño detallado, programadores, responsables de las bases de datos, responsables de la verificación y pruebas y responsables del soporte técnico.
- Asigna el trabajo a los miembros del equipo de desarrollo.
- Atiende las necesidades de los miembros del equipo de desarrollo, brindando solución oportuna a las mismas.
- Establece la planeación del proyecto de desarrollo y el control de los avances, siendo el responsable del cumplimiento de dicha planeación, y, por lo tanto, de llevar a buen término el proyecto de desarrollo de software.

## II.4 Los analistas de sistemas

Como parte de su rol en el equipo de desarrollo, el analista de sistemas debe ejercer las siguientes funciones:

- Interpretar las necesidades y expectativas del cliente.
- Determinación de los requerimientos funcionales y no funcionales del sistema software.
- Análisis de los requerimientos funcionales y no funcionales del sistema software.
- Construcción de prototipos del sistema software.
- Especificación de los requerimientos funcionales y no funcionales del sistema software.

## II.5 Los arquitectos de software

Como parte de su rol en el equipo de desarrollo, el arquitecto de software debe ejercer las siguientes funciones:

- Definición de la arquitectura lógica del software.
- Definición de la arquitectura física del software.
- Gestión de los requerimientos no funcionales.
- Selección de la tecnología y manejo de los riesgos tecnológicos.
- Mejora continua de la arquitectura del software (tanto lógica como física).
- Sugerir y colaborar en la implementación de la arquitectura lógica del software, asegurando que todos los aspectos de la arquitectura se estén implementando de forma correcta.

## II.6 Los encargados del diseño detallado

Como parte de su rol en el equipo de desarrollo, el encargado del diseño detallado debe ejercer las siguientes funciones:

- Partiendo de la propuesta del diseño arquitectónico, modelar en detalle cada una de las partes o componentes (módulos, clases, funciones,

procedimientos, etc.) que integran la arquitectura propuesta, aplicando un diseño modular efectivo.

- Diseñar de forma detallada, a través de los modelos más apropiados, la comunicación y dependencia entre los módulos diseñados.
- Verificar que cada uno de los componentes diseñados a detalle satisface los requerimientos funcionales especificados en la etapa de análisis, y, por lo tanto, las necesidades y expectativas del cliente.
- Proporcionar una especificación de diseño a un nivel de detalle tal, que el proyecto quede listo para la fase de implementación/codificación.

## II.7 Los programadores

Como parte de su rol en el equipo de desarrollo, el programador debe ejercer las siguientes funciones:

- Traducir la propuesta del diseño detallado (módulos, clases, funciones, procedimientos, etc.) que integran la arquitectura propuesta, a un lenguaje de programación (C++, Java, Python, entre otros.).
- Garantizar que la codificación sea una consecuencia natural del diseño, convirtiendo cada componente del diseño detallado en un programa en el lenguaje de programación seleccionado.
- Verificar que cada uno de los componentes implementados satisface los requerimientos funcionales especificados en la etapa de análisis, y, por lo tanto, las necesidades y expectativas del cliente.

## II.8 Los Responsables de las Bases de Datos

Como parte de su rol en el equipo de desarrollo, el responsable de la base de datos debe ejercer las siguientes funciones:

- Seleccionar el modelo de datos (red, jerárquico, entidad-relación, relacional, orientado a objetos, por ejemplo.) o sistema manejador de bases de datos más apropiado a las características y exigencias del sistema de software.
- Efectuar el análisis y diseño de la base de datos, atendiendo al modelo de datos seleccionado.
- Desarrollar el modelo físico e implementación de la base de datos.
- Efectuar la generación de datos de pruebas.

## II.9 Los responsables de la verificación y pruebas

Como parte de su rol en el equipo de desarrollo, el responsable de la verificación y pruebas debe ejercer las siguientes funciones:

- Aplicar las pruebas del software, como un conjunto de actividades encaminadas a identificar posibles fallas, errores, omisiones, etc., tanto durante el desarrollo del producto software como durante la ejecución del mismo.
- Garantizar que las pruebas del software abarquen cada una de las fases de desarrollo del producto software, y no se centren solamente en la ejecución del software, ya que en cada una de éstas es posible detectar fallas, errores, omisiones, etc.
- Aplicar las pruebas del software relacionadas con la recolección de los requerimientos, el análisis de los requerimientos, el diseño arquitectónico, el diseño detallado y la codificación.

## II.10 Los responsables de soporte técnico

Como parte de su rol en el equipo de desarrollo, el responsable de soporte técnico debe ejercer las siguientes funciones:

- Configuración y mantenimiento de toda la infraestructura de cómputo – incluyendo, redes de computadoras, servidores de aplicaciones, gestores de bases de datos, entornos de desarrollo integrados, entre otras configuraciones– requeridos para el desarrollo del sistema de software.
- Proporcionar asistencia a los restantes miembros del equipo de desarrollo ante cualquier problema o dificultad que surja en el uso de cualquier producto o servicio tanto en hardware como en software.

## II.11 Conformación de los equipos de desarrollo en proyectos de software a gran escala

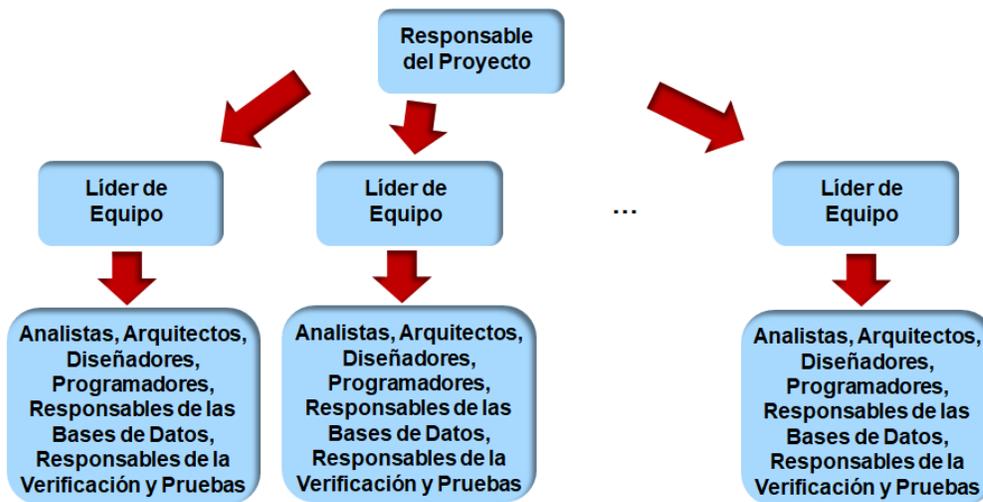
Comúnmente, el tamaño y complejidad del sistema de software a gran escala a desarrollar exige el trabajo de dos o más equipos de desarrollo. Cuando ese es el caso, entonces la conformación de dichos equipos sigue una de dos variantes:

- Variante 1: Equipos de desarrollo de software con los mismos roles y responsabilidades, donde cada equipo comúnmente se encarga del desarrollo de uno o más subsistemas, módulos o componentes del sistema de software a gran escala (ver Figura. 2.2).
- Variante 2: Equipos de desarrollo de software especializados en actividades específicas del desarrollo de software. En esta variante, cada

equipo juega un rol o responsabilidad específico durante todo el proceso de desarrollo de software (ver Figura 2.3). Por ejemplo, equipo especializado en la recolección y análisis de los requerimientos, equipo especializado en la arquitectura del sistema, equipo especializado en el diseño detallado de los componentes del sistema.

Si bien es cierto que en ocasiones todo el proceso de desarrollo de un sistema de software a gran escala recae sobre un único equipo de desarrollo, conformado según la variante 1; lo deseable sería contar con dos o más equipos de desarrollo, dependiendo de qué tan complejo y grande sea el sistema de software a desarrollar.

Por otra parte, las complejas características del dominio del problema podrían conllevar a la conformación de los equipos de desarrollo de software según la variante 1, considerando además que cada equipo posee experiencia en un determinado rasgo o característica particular del dominio del problema, lo cual exige el dominio de particulares técnicas o tecnologías por parte de los miembros del equipo. Esta situación se ilustrará en el próximo epígrafe, tomando nuevamente como ejemplo el Subsistema de Procesamiento Integral del Sistema de Detección de Fugas y Tomas Clandestinas en Ductos que Transportan Hidrocarburos.



**Figura2.2.** Organigrama de los roles y responsabilidades: variante 1 (equipos con las mismas responsabilidades y capacidades).



**Figura 2.3.** Organigrama de los roles y responsabilidades: variante 2 (especialización del trabajo de desarrollo).

## II.12 Caso de Estudio

### II.12.1 Sistema de Detección de Fugas en Ductos que Transportan Hidrocarburos.

#### II.12.1.1 Equipos de Desarrollo, Roles y Responsabilidades.

En el Epígrafe 1.6 ya nos referimos al alcance y principales características del Subsistema de Procesamiento Integral de Datos (SPI) del Sistema de Detección de Fugas en Ductos que Transportan Hidrocarburos [5, 6], como ejemplo de sistema de software a gran escala. En este epígrafe nos referiremos a las características del subsistema SPI que conllevaron a la conformación de los equipos de desarrollo, y, por lo tanto, a la variante seleccionada.

Regresando a la Tabla 1.2 del capítulo I, en la característica denominada “Funcionalidad extensa y compleja” (primera fila de la tabla) podemos identificar las tres grandes funcionalidades del subsistema SPI:

- 1) Comunicación síncrona con los sistemas Lazo Cerrado (LC), Vigilantes Virtuales (VV), Acústica (AC), Fibra Óptica (FO), y con el sistema de control y monitoreo a nivel superior.
- 2) Conformación y lectura del patrón integral de datos, a partir de los eventos reportados por los sistemas LC, VV, AC y FO.
- 3) Ejecución de los diagnósticos particulares y diagnóstico integral, basados en técnicas de inteligencia artificial, sobre el patrón integral de datos conformado.

La complejidad de estas tres funcionalidades a nivel superior (módulos), y la dependencia de cada una de éstas de conocimientos y habilidades en modelos,

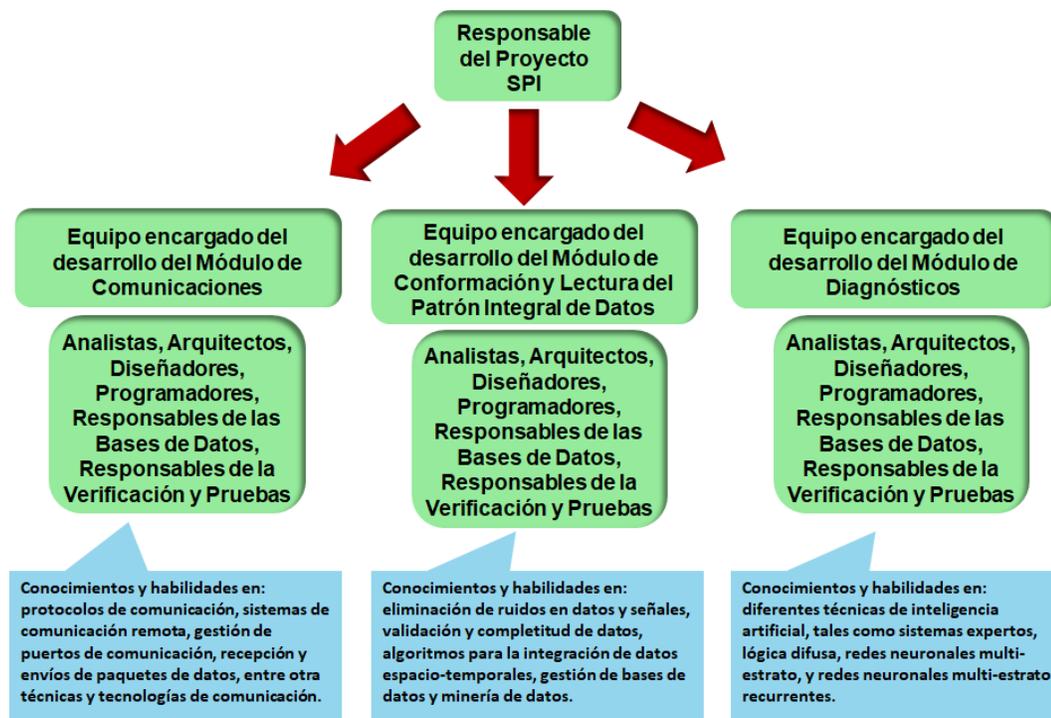
técnicas y tecnologías muy particulares y específicos, hace que emerja la propuesta de considerar tres equipos de desarrollo de software, conformados según la variante 1 (ver Figura 2.2). Además, esta variante se complementa con el hecho de que cada equipo de desarrollo debe poseer conocimientos y experiencia en el manejo de los modelos, técnicas y tecnologías requeridos para el desarrollo del módulo asignado.

El equipo de trabajo encargado del desarrollo del módulo para la comunicación síncrona con los subsistemas remotos de detección, y con el sistema de control y monitoreo a nivel superior, debe poseer conocimientos y habilidades en: protocolos de comunicación, sistemas de comunicación remota, gestión de puertos de comunicación, recepción y envíos de paquetes de datos, entre otra técnicas y tecnologías de comunicación.

Por su parte, el equipo de trabajo que tiene a su cargo el módulo de conformación y lectura del patrón integral de datos, a partir de los eventos reportados por los subsistemas remotos de detección, requiere conocimientos y habilidades en: eliminación de ruidos en datos y señales, validación y completitud de datos, algoritmos para la integración de datos espacio-temporales, gestión de bases de datos y minería de datos.

Finalmente, el equipo de trabajo encargado de la ejecución de los diagnósticos particulares y diagnóstico integral sobre el patrón integral de datos conformado, requiere de conocimientos y habilidades en diferentes técnicas de inteligencia artificial, tales como sistemas expertos, lógica difusa, redes neuronales multiestrato, y redes neuronales multiestrato recurrentes.

Tomando en consideración las motivaciones antes expuestas, la Figura 2.4 ilustra la conformación de los equipos de trabajo encargados del desarrollo del subsistema SPI.



**Figura 2.4.** Conformación de los equipos de trabajo para el desarrollo del subsistema SPI.

## II.13 Referencias del capítulo

1. Screerer, A. Coordination in Large-Scale Agile Software Development: Integrating Conditions and Configurations in Multiteam Systems (Progress in IS). Springer, 2017.
2. Pfleeger, S. L. Ingeniería de software: Teoría y práctica. Pearson Education, 2002.
3. Pressman, R. S. Ingeniería del software: Un enfoque práctico. McGraw-Hill, 2010.
4. Sommerville, I. Ingeniería del software. Pearson Addison Wesley, 2012.
5. González-Pérez, P.P., Sadovnychyy, A., Alarcón Ramos, L.A., González González, D., Elorza Gómez, K., Peña Falcón, J.L. Sistema para la detección de fugas y tomas clandestinas en ductos de transporte de gas y líquidos. Sistema Computacional de Procesamiento Integral de Datos (PI). Documento de Análisis y Diseño. Reporte Técnico. 2013.
6. González-Pérez, P.P., Schaum, A., Sadovnychyy, A., Bernal Jaquez, R., Méndez Gurrola, I.I., Alarcón Ramos, L.A., Rosales Cruz, L. Sistema para la detección de fugas y tomas clandestinas en ductos de transporte de gas y líquidos. Sistema Computacional de Procesamiento Integral de Datos (PI). Estado del Arte. Reporte Técnico. 2011.

# Capítulo III El proceso de Requerimientos

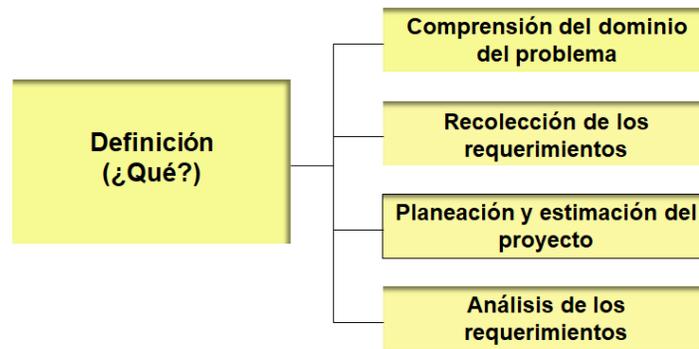
## III.1 El alcance del análisis de requerimientos

Como ya mencionamos en el Capítulo I, una de las principales características de los sistemas de software a gran escala es la funcionalidad extensa y compleja que se requiere de los mismos, así como la dependencia de una gran variedad de requerimientos no funcionales, tanto aquellos que caracterizan el funcionamiento global del sistema software como aquellos que caracterizan al sistema software como producto [1-6]. De aquí la gran importancia del proceso de requerimientos [7, 8] durante el desarrollo de este tipo de sistemas, y de forma muy especial la validación/verificación de los requerimientos y la gestión de los riesgos atribuibles a los mismos en una etapa temprana del proceso de desarrollo [9-13].

Durante el proceso de requerimientos se hace énfasis “lo que hará” el sistema software en lugar de concentrarse en el “cómo lo hará”. Durante la extracción y análisis de requerimientos se deben comprender las necesidades o solicitudes del usuario y traducirlas a funcionalidades o prestaciones que caracterizarán al sistema software que se va a desarrollar. Estas necesidades o solicitudes del usuario deben ser recopiladas, especificadas, modeladas, refinadas y verificadas.

Como se ilustra en la Figura. 3.1, durante el proceso de requerimientos:

- Se establece la razón de ser del proyecto y se determina su alcance.
- Se adquiere una visión aproximada del proyecto en términos de los principales requerimientos funcionales y no funcionales (restricciones).
- Se desarrolla un primer análisis del dominio del problema.
- Se efectúan las primeras estimaciones, aún no precisas, del proyecto, en cuanto a tamaño del mismo en líneas de código o puntos de función, esfuerzo personas-mes y planeación en meses.

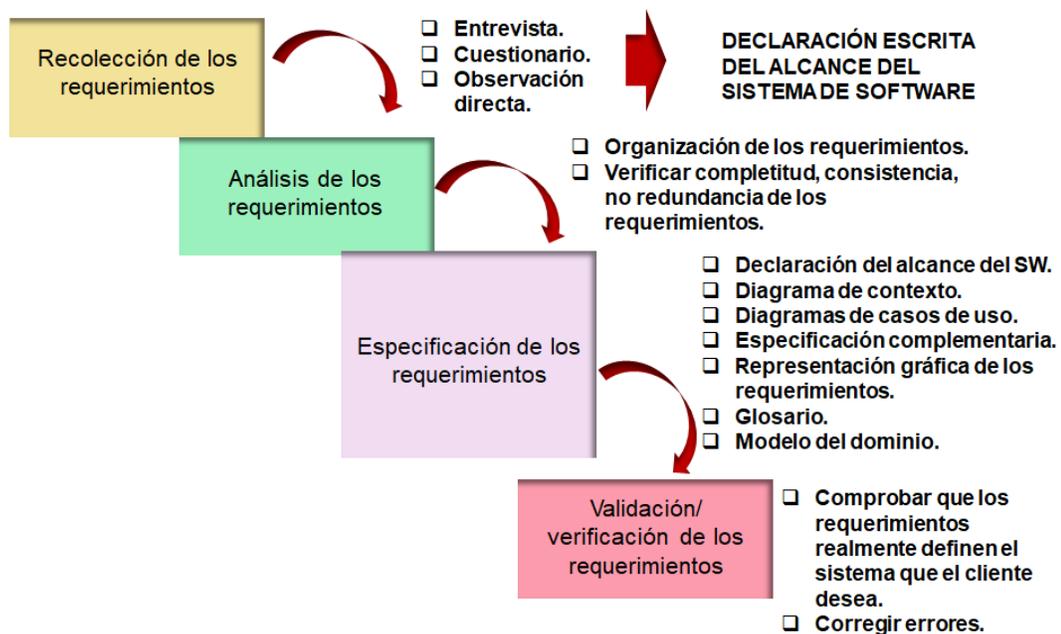


**Figura 3.1.** Alcance del proceso de requerimientos.

### III.2 Actividades del proceso de requerimientos

El proceso de requerimientos comúnmente abarca las siguientes actividades [7, 8], tal como se ilustra en la Figura 3.2:

- Recolección de los requerimientos.
- Análisis de los requerimientos.
- Especificación de los requerimientos.
- Validación/verificación de los requerimientos.



**Figura 3.2.** Etapas del Proceso de Requerimientos.

### III.2.1 Recolección de requerimientos

La recolección (captura o extracción) de los requerimientos [7-11] tiene como objetivo principal la comprensión de aquello que los clientes y los usuarios esperan que haga el sistema. Durante la recolección de los requerimientos se identifican los aspectos clave que el sistema requiere y se descartan los aspectos irrelevantes. Entre las técnicas comúnmente utilizadas durante la recolección de los requerimientos se encuentran:

- La entrevista con el cliente y los usuarios.
- La observación de tareas, en la que se revelan problemas, detalles, estrategias y estructuras de trabajo que son difíciles de captar claramente con las entrevistas.
- El prototipado, técnica con la cual se va modificando un prototipo hasta que éste cumple con las expectativas del cliente.

Para ilustrar alguna de las técnicas utilizadas durante la recolección de requerimientos, las Tablas 3.1 a la 3.3 relacionan fragmentos de la entrevista efectuada, durante la homónima fase de desarrollo del sistema web Club Deportivo Virtual Sport Planet, un sistema web a gran escala desarrollado a nivel prototipo por uno de los autores de este material.

**Tabla 3.1.** Entrevista efectuada durante la recolección de requerimientos del sistema web Club Deportivo Virtual – Fragmento 1.

Pregunta	Respuesta
<b>Preguntas generales</b>	
1. ¿Cuál es el objetivo del desarrollo del software?	Automatizar e integrar toda una gama de servicios, que actualmente se ejecutan de forma manual o semiautomatizada, en un único sistema o aplicación web, cuyos principales usuarios sean los gerentes, los entrenadores, los socios y el público en general. Una las principales prioridades del sistema es permitir que los socios y público en general puedan efectuar vía Web pagos tales como membresías, cuotas de mantenimiento, cursos y otras actividades, con cargo a tarjeta de crédito y débito.
2. ¿De las actividades que se desean integrar como prestaciones del sistema web hay algunas ya automatizadas?	Se cuentan con archivos en Excel y algunas tablas en MySQL para la gestión (altas, bajas, actualización, seguimiento, consulta, etc.) de la información de los entrenadores y socios.
3. ¿Qué otras actividades que actualmente se desarrollan de forma manual o semiautomatizada se desean integrar en el sistema web?	<p>Por ejemplo, la gestión de las rutinas de ejercicios y programas de ejercicios cardiovascular, lo cual es de gran importancia para los entrenadores. Actualmente, ellos efectúan toda la gestión (creación, actualización, seguimiento, etc.) de manera manual, lo cual ocasiona múltiples problemas, como pérdidas de los folletos, poca actualización, pobre seguimiento, etc.</p> <p>Por otra parte, sería muy importante que toda la información (comunicados, promociones, eventos deportivos, tips, etc.) dirigida a los entrenadores, socios y público en general, se haga a través de un sitio web.</p>
4. ¿Qué otras características se desean que exhiba el sistema web?	Ante todo, que sea muy agradable a la vista, amigable, muy interactivo, que el usuario se sienta muy cómodo y a gusto con su uso. Por otra parte, que sea seguro y confiable en el manejo de toda la información que gestiona, sobretodo, brindando una gran robustez en las transacciones de pagos.

**Tabla 3.2.** Entrevista efectuada durante la recolección de requerimientos del sistema web Club Deportivo Virtual – Fragmento 2.

Pregunta	Respuesta
<b>Preguntas orientadas a cada actividad a automatizar</b>	
1. ¿En qué consiste la actividad o tarea a automatizar?	GESTIÓN DE PAGOS. Que los socios y público en general puedan efectuar los pagos de membresías, mantenimiento, inscripción a cursos, participación en eventos especiales, etc., a través del sistema web, con cargo a tarjetas de crédito y débito.
2. ¿Qué tan prioritaria es?	Prioridad alta. Se necesita que sea una de las principales tareas automatizada e integrada en el sistema web.
3. ¿Qué información de entrada requiere?, ¿cuál es la fuente de dicha información?	Tipo y monto del servicio a pagar, datos personales, datos de la tarjeta de crédito o débito, etc. La fuente que origina dicha información es el socio o público en general.
4. ¿Qué acción/información de salida produce?, ¿cuál es el formato requerido de la salida?	Pago efectuado, comprobante de pago, registro en la BD de Pagos. El comprobante de pago podría tener el mismo formato que cualquier comprobante bancario. Por otra parte, en la tabla de Pagos será necesario registrar toda la información de la transacción efectuado, una vez recibida la aprobación bancaria.
5. ¿De cuáles otras operaciones básicas o primitivas depende dicha actividad?	La gestión de pago se puede ver como una secuencia de actividades: entrada de los datos de los pagos a efectuar, validación de dicha información, solicitud de la autorización de pago, y, en su caso, registro del pago en la BD de Pagos.
6. ¿Qué otras características se asocian a esta actividad?	Seguridad, robustez, confiabilidad.

**Tabla 3.3.** Entrevista efectuada durante la recolección de requerimientos del sistema web Club Deportivo Virtual – Fragmento 3.

Pregunta	Respuesta
<b>Preguntas orientadas a cada actividad a automatizar</b>	
1. ¿En qué consiste la actividad o tarea a automatizar?	GESTIÓN DE RUTINAS. Que los Entrenadores Personales y Entrenadores de Piso puedan crear, actualizar y dar seguimiento a las rutinas de forma automatizada.
2. ¿Qué tan prioritaria es?	ALTA.
3. ¿Qué información de entrada requiere?, ¿cuál es la fuente de dicha información?	Lista de ejercicios:  Para cada ejercicio se requiere proporcionar: tipo de ejercicio, nombre del ejercicio, series, repeticiones, peso, descanso, etc. La fuente que proporciona dicha información es el Entrenador.
4. ¿Qué acción/información de salida produce?, ¿cuál es el formato requerido de la salida?	Registro y visualización del formato de la rutina en forma de tabla, con un encabezado para los datos personales del socio, y a continuación una segunda sección para las columnas Tipo de Ejercicio, Nombre del Ejercicio, Series, Repeticiones, Peso, Descanso. Finalmente, una sección para observaciones y seguimiento.
5. ¿De cuáles otras operaciones básicas o primitivas depende dicha actividad?	Una vez creada la rutina, las actividades básicas que se pueden ejecutar son actualización y seguimiento de la rutina.
6. ¿Qué otras características se asocian a esta actividad?	Altamente visual, gráfica e interactiva.

### III.2.2 Análisis de requerimientos

Durante el análisis de requerimientos [3] se extraen las características operacionales del software, se indican las interfaces entre usuario y sistema y se establecen las restricciones que debe cumplir el software. También se traducen los requisitos del usuario a requerimientos de software y se clasifican los requerimientos. Además, se detecta si hay conflictos entre los requerimientos y se resuelven estos conflictos.

### III.2.3 Especificación de requerimientos

La especificación y modelado de los requerimientos [7-11] comúnmente se apoyan en una gama de técnicas y métodos que proporcionan una mayor comprensión y claridad de los mismos, garantizando así la continuidad del ulterior proceso de desarrollo. A través de la especificación de los requerimientos se construyen diferentes artefactos, modelos y técnicas que muestran las propiedades más importantes del dominio del problema, tales como:

- Actores.
- Dependencias de otros sistemas, subsistemas, bases de datos, etc.
- Requerimientos funcionales.
- Requerimientos no funcionales o restricciones.
- Entidades.
- Relaciones entre entidades.

Se han propuesto diferentes técnicas y modelos para la especificación de los requerimientos de acuerdo al enfoque de desarrollo del software [9-11], en este material enfatizaremos sobre el uso de seis artefactos, los cuales han demostrado ser de gran utilidad en la especificación de requerimientos durante el desarrollo de software a gran escala:

- Declaración del alcance del sistema de software.
- Modelo contextual.
- Diagrama de casos de uso.
- Especificación complementaria.
- Glosario de entidades.
- Modelo del dominio.

La Figura 3.3 ilustra la secuencia de aplicación de tales artefactos durante la actividad especificación de los requerimientos, mientras que la Tabla 3.4 ofrece una descripción de los mismos. Los elementos que integran la especificación de los requerimientos serán conceptos del dominio de la aplicación.

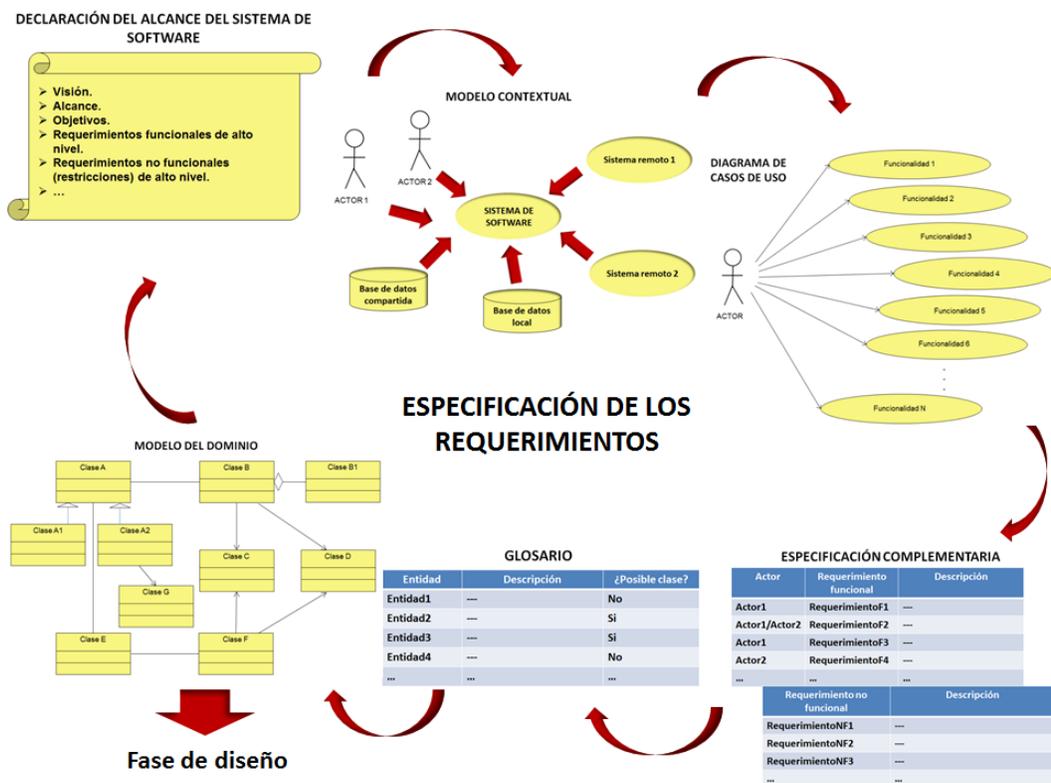


Figura 3.3. Secuencia de aplicación de los artefactos durante la especificación de los requerimientos.

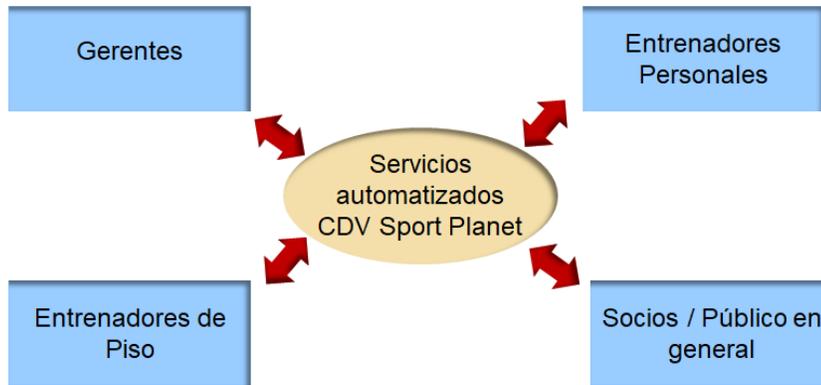
Tabla 3.4. Artefactos utilizados durante la especificación de los requerimientos.

Artefacto, modelo o técnica	Descripción
Declaración del alcance del sistema de software	<p>Describe los objetivos y las restricciones de alto nivel. Brinda una descripción del dominio desde una perspectiva de las principales funcionalidades/requerimientos y objetos identificados. Proporciona elementos para la toma de decisiones, e incluye, entre otros elementos:</p> <ul style="list-style-type: none"> <li>➤ <b>Visión.</b> Describe la visión del proyecto, planteando de forma clara que técnica / enfoque / paradigma, etc. será propuesto y que problema será resuelto. La visión se complementa con un diagrama de contexto del sistema.</li> <li>➤ <b>Objetivos.</b> Describe los objetivos del proyecto: ¿qué se pretende hacer?, ¿qué problema será resuelto?, ¿cuáles son las principales funcionalidades que debe satisfacer</li> </ul>

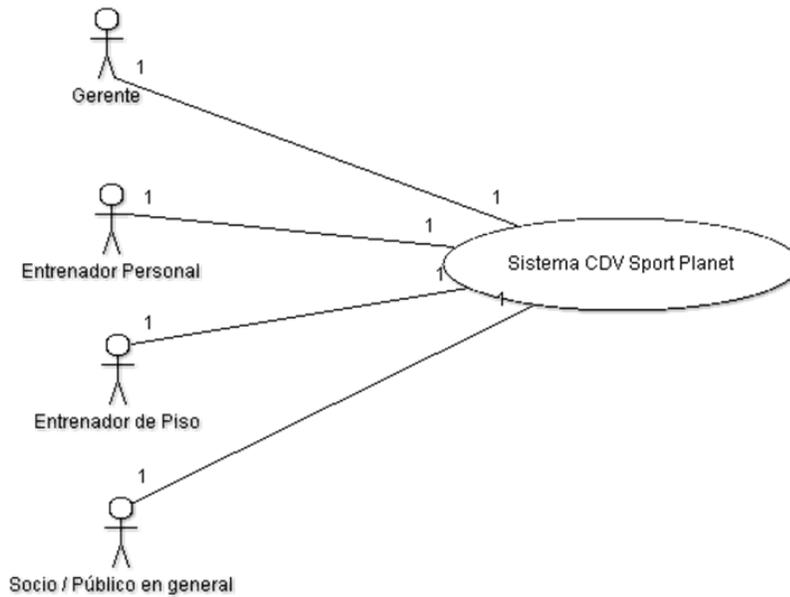
	<p>el sistema de software?, ¿qué resultados se alcanzarán?, etc.</p> <ul style="list-style-type: none"> <li>➤ <b>Restricciones de alto nivel.</b> Principales restricciones (requerimientos no funcionales) que afectan el desarrollo del proyecto y ejecución del sistema. Ejemplos de restricciones son las siguientes: arquitectura, interacción con otros sistemas, seguridad, tolerancia a fallas, tipos de usuario, etc.</li> </ul>
<p><b>Modelo contextual</b></p>	<p>Ilustra los principales componentes del sistema de software, sus dependencias de otros sistemas o entidades, así como los límites entre el sistema de software y su ambiente. Entre las principales dependencias del sistema de software se encuentran las siguientes:</p> <ul style="list-style-type: none"> <li>➤ Sistemas o componentes remotos.</li> <li>➤ Bases de datos compartidas.</li> <li>➤ Bases de datos propias del sistema de software.</li> <li>➤ Otros dispositivos de hardware requeridos, tales como sensores, lectores ópticos, actuadores, etc.</li> </ul>
<p><b>Diagrama de casos de uso</b></p>	<p>Es un mecanismo que permite capturar, hacer visibles y comprensibles los objetivos y requerimientos del sistema. Los casos de uso describen los requerimientos funcionales del sistema y su relación con los requerimientos no funcionales. Los casos de uso son requerimientos funcionales que indican qué hará el sistema.</p> <p>Un caso de uso se define en términos de <b>actores</b> y <b>escenarios</b>.</p> <ul style="list-style-type: none"> <li>➤ <b>Un actor</b> representa cualquier tipo de entidad externa caracterizada de un comportamiento y que interactúa con el sistema. Ejemplos de actores son: una persona identificada por un rol (operador, usuario, gerente, administrador, etc.), un departamento, una organización, otro sistema, etc.</li> <li>➤ <b>Un escenario</b> es una secuencia determinada de acciones e interacciones entre los actores y el sistema en cuestión (a través de los requerimientos que caracterizan la funcionalidad de dicho sistema).</li> </ul>

<p><b>Especificación complementaria</b></p>	<p>Es la descripción de todos los requerimientos funcionales y no funcionales, la cual complementa el modelo de casos de uso. Esta descripción puede ser hecha de forma textual, con tablas, con otros diagramas, etc. Además de los requerimientos funcionales, entre los elementos de la especificación complementaria se pueden encontrar:</p> <ul style="list-style-type: none"> <li>➤ Requerimientos de calidad del sistema.</li> <li>➤ Requerimientos sobre la funcionalidad, facilidad de uso, fiabilidad, rendimiento y soporte.</li> <li>➤ Restricciones de software y hardware (sistemas operativos, infraestructura de red, dispositivos de entrada/salida de datos, etc.).</li> <li>➤ Restricciones de desarrollo (herramientas, lenguajes, etc.).</li> <li>➤ Documentación.</li> <li>➤ Entorno físico.</li> </ul>
<p><b>Glosario de entidades</b></p>	<p>Se refiere a la terminología clave del dominio. Es la definición de los términos más importantes que sobresalen en el análisis del documento de los requerimientos. Estos términos no tienen que corresponder necesariamente con los objetos del dominio. El glosario se puede construir como una lista de los términos relevantes del dominio y sus definiciones.</p> <p>El glosario no debe recoger todos los posibles términos, sino aquellos que no están claramente definidos, aquellos que pudieran resultar ambiguos, o aquellos otros que por su relevancia e importancia en el dominio del problema deben quedar claramente definidos.</p>
<p><b>Modelo del dominio</b></p>	<p>El modelo del dominio se expresa como un diagrama de clases que describe los tipos de entidades (clases) que componen el sistema y los diferentes tipos de relaciones estáticas que existen entre éstas. Las clases del modelo del dominio deben provenir del glosario previamente desarrollado, incluyendo sólo aquellas entidades que representan aspectos relevantes de la lógica del sistema.</p>

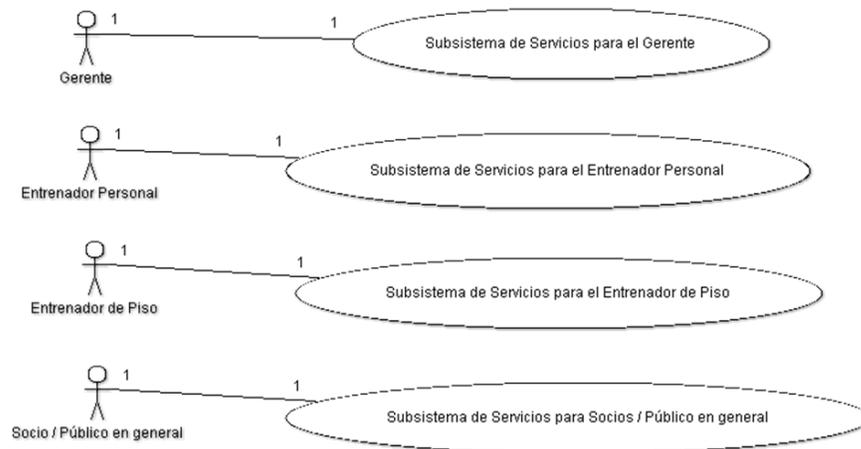
Las Figuras 3.4 a la 3.8 ilustran algunas de las técnicas utilizadas durante la especificación de los requerimientos del sistema web Club Deportivo Virtual.



**Figura 3.4.** Especificación de los requerimientos del sistema web Club Deportivo Virtual – Modelo de contexto.



**Figura 3.5.** Especificación de los requerimientos del sistema web Club Deportivo Virtual – Modelo de casos de uso.



**Figura 3.6.** Especificación de los requerimientos del sistema web Club Deportivo Virtual – Modelo de casos de uso.

Actor	Funcionalidad requerida
<b>Gerente</b>	<input type="checkbox"/> Gestión de pagos. <input type="checkbox"/> Gestión de cursos. <input type="checkbox"/> Gestión de la información publicada. <input type="checkbox"/> Gestión de socios. <input type="checkbox"/> Gestión de entrenadores. <input type="checkbox"/> Gestión de rutinas.
Actor	Funcionalidad requerida
<b>Entrenador Personal</b>	<input type="checkbox"/> Consulta de información publicada por el Club Deportivo. <input type="checkbox"/> Gestión de rutinas de entrenamiento personalizado. <input type="checkbox"/> Gestión de programas de entrenamiento cardiovascular <input type="checkbox"/> Consulta de información de los socios.

**Figura 3.7.** Especificación de los requerimientos del sistema web Club Deportivo Virtual – Fragmento de la Especificación Complementaria.

Entidad	Descripción
Club Deportivo	Entidad que proporciona un espacio, infraestructura y servicios dedicados al ejercicio físico, práctica de deportes, entrenamientos personalizados y otra gama de actividades relacionadas con el bienestar físico.
Gerente	Ejecutivo del Club Deportivo que goza de determinadas prerrogativas y ejecuta tareas de dirección y control que involucra entrenadores y socios.
Entrenador Personal	Entrenador que brinda atención personalizada (entrenamiento) a socios del club, a través del diseño de rutinas, asesoramiento y guía en la ejecución de la rutina, seguimiento y cambios de la rutina, etc.
Entrenador de Piso	Entrenador que proporciona información sobre el uso de la infraestructura del club, indica rutinas preliminares de entrenamiento generalizado, etc.

**Figura 3.8.** Especificación de los requerimientos del sistema web Club Deportivo Virtual – Fragmento del Glosario.

### III.2.4 Validación/verificación de requerimientos

La validación de los requerimientos [7-11] tiene como objetivo detectar defectos en los requerimientos antes de invertir recursos en las siguientes etapas del proceso de desarrollo. El objetivo principal de la validación de los requerimientos es comprobar que éstos realmente definen el sistema que el cliente desea y que lo que describen es lo que el cliente pretende ver en el producto final. Otro objetivo importante de la validación es asegurar que los requerimientos estén completos, sean exactos y consistentes.

Las principales técnicas de validación de requerimientos son:

- Revisiones de requerimientos.
- Construcción de prototipos.

Por otra parte, la verificación de los requerimientos pretende asegurar que el sistema de software podrá satisfacer las necesidades del cliente y se lleva a cabo mediante las pruebas de aceptación. Las *pruebas de aceptación* sirven para convencer al cliente de que el sistema cumple con lo que él pidió. Debe diseñarse por lo menos una prueba para cada requerimiento. Si un requerimiento no se puede probar significa que el requerimiento está mal hecho y es necesario replantearlo.

### III.3 Casos de estudio

Al igual que en los capítulos I y II, en este epígrafe retomaremos el Subsistema de Procesamiento Integral de Datos, del Sistema de Detección de Fugas y Tomas Clandestinas en Ductos que Transportan Hidrocarburos [14, 15], para ilustrar algunas técnicas utilizadas durante la fase análisis de requerimientos, en particular la especificación de requerimientos. De forma adicional, introduciremos un segundo caso de estudio, la plataforma bioinformática Evolution [16], el cual nos permitirá

ilustrar aspectos relevantes de las diferentes fases del proceso de desarrollo de software, tanto en este capítulo como a través del resto del material.

### III.3.1 Subsistema de Procesamiento Integral de Datos para la Detección de Fugas en Ductos que Transportan Hidrocarburos.

#### *Declaración del alcance del subsistema SPI: objetivos, visión y alcance*

El Subsistema de Procesamiento Integral de Datos (SPI) tiene como principal objetivo aumentar la efectividad en la detección oportuna de fugas y tomas clandestinas en los ductos de hidrocarburo de PEMEX. Para lograr lo anterior, el subsistema SPI incorpora en su procesamiento integral dos importantes estrategias:

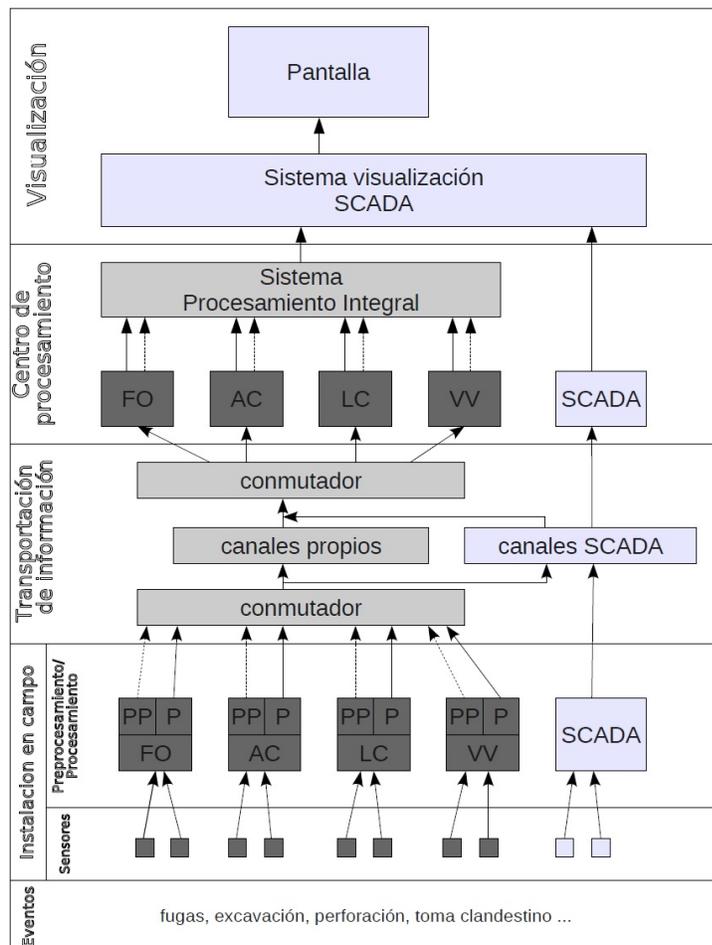
- iii. Integración de las salidas producidas por los cuatro subsistemas para la detección de fugas y tomas clandestinas (a los cuales hemos hecho referencia en los epígrafes anteriores): sistema de cálculo de balance de masa con métodos no lineales de estimación en Lazo Cerrado (LC), sistema de cálculo de balance con uso del Método de Vigilantes Virtuales (VV), sistema basado en Acústica (AC) y sistema basado en fibra óptica (FO).
- iv. Integración de un grupo de valiosas técnicas de inteligencia artificial, tales como sistemas expertos, redes neuronales artificiales multiestrato, redes neuronales artificiales recurrentes y mapas cognitivos difusos; para el análisis y clasificación de los eventos causados por fugas y tomas clandestinas. Tales eventos corresponden a las salidas de los cuatro sistemas previamente mencionados, las cuales integradas constituyen el conjunto de datos de entrada al subsistema SPI.

Como se puede apreciar en la Figura 3.9, el contexto y alcance en el cual debe operar el subsistema SPI queda completamente definido por los cinco sistemas que se relacionan a continuación:

- Un sistema de Control y Monitoreo a Nivel Superior.
- Dos sistemas de detección de fugas basados en el cálculo de balance de masa.
- Un sistema basado en acústica.
- Un sistema basado en fibra óptica.

Cada evento que ocurre en un ducto cambia las características del medio ambiente: las fugas producen sonido cuando el líquido o gas sale de la tubería, contaminan la tierra circundante del ducto, aumentan la temperatura; la excavación y perforación de tuberías generan ruido singular; las fugas y tomas clandestinas disminuyen la presión en el ducto y cambian las características del flujo, etc. Como se puede apreciar en la Figura 3.9, cada cambio de estas características puede ser detectado con el uso de diferentes tipos de sensores. El sistema de sensores en base a fibra óptica detecta líquidos y gas en tierra, sonidos, movimientos de tubería y cambios de temperatura. El sistema basado en acústica detecta diferentes ruidos acústicos. Los sistemas basados en el cálculo de balance de masa y el sistema de

control y monitoreo a nivel superior, detectan cambios en las características del flujo dentro de la tubería, tales como temperatura, presión, viscosidad, etc.



**Figura 3.9.** Modelo de contexto del Subsistema de Procesamiento Integral de Datos.

Cada uno de estos cinco sistemas trabaja de forma independiente y puede detectar solo algunas características del evento. Además, la certidumbre de detección de cada sistema depende de diferentes parámetros de explotación. Para evitar los falsos positivos de cada uno de estos sistemas y para mejorar la eficiencia de la detección de eventos, se propone el desarrollo del subsistema de procesamiento integral de datos SPI. La idea fundamental del subsistema SPI es la siguiente: tomando en consideración los parámetros que caracterizan el ducto y las salidas producidas por cada uno de los cuatro sistemas subsistemas remotos, prevenir falsos positivos y detectar eventos en ductos con una mayor precisión y eficiencia.

Cuando ocurre un evento, cambian las características del ambiente interno y externo al ducto, y los sistemas remotos de detección de fugas, en base a dichos cambios, logran la detección de dicho evento. Cada sistema puede usar sus propios sensores, como es el caso de los sistemas basados en fibra óptica y acústica, o

compartirlos, como ocurre entre los sistemas basados en balance de masa. Todos los sensores se localizan cerca del ducto y sus datos son enviados al centro de pre-procesamiento (PP) o procesamiento (P) (ver Figura 3.9), dependiendo de donde haya sido instalado el sistema.

Un centro de pre-procesamiento es un lugar especial con líneas de alimentación y comunicación. En los centros PP, los datos provenientes de los sensores se procesan en una computadora estándar o con sistemas basados en procesadores especiales. Los centros PP están sobre la vía del ducto a una distancia de varios kilómetros uno del otro.

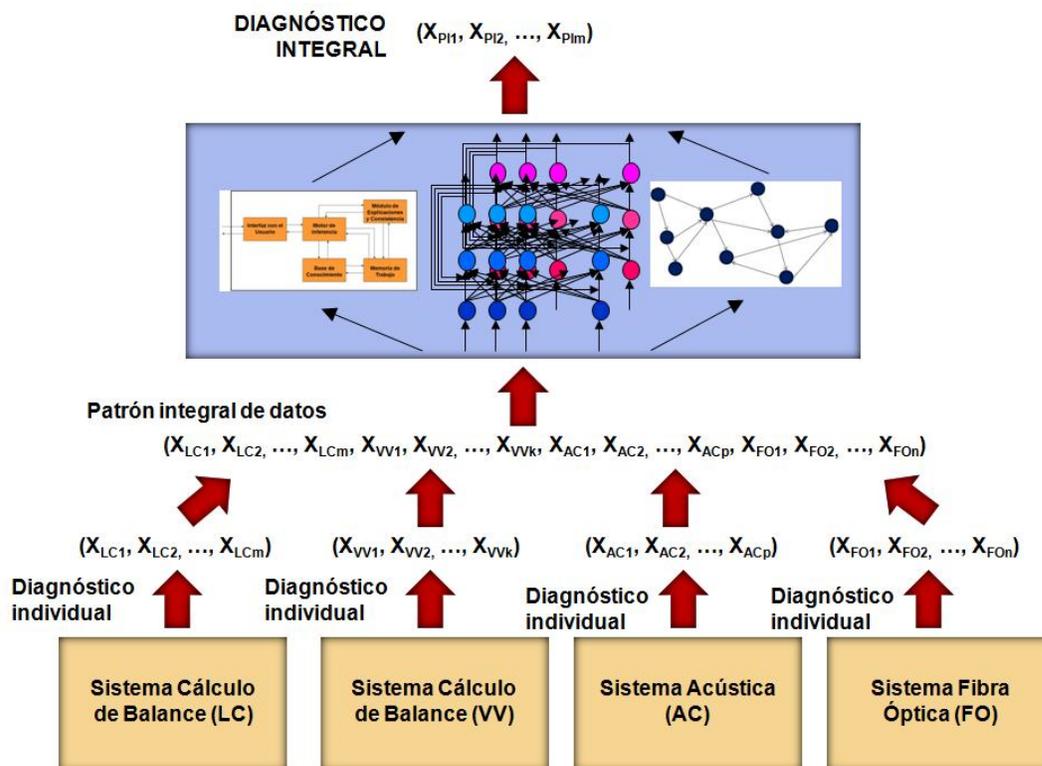
Después del pre-procesamiento los datos llegan al conmutador que los envía al sistema de control y monitoreo a nivel superior. El conmutador puede enviar los datos a través de sus propios canales de comunicación o puede insertarlos en el canal de dicho sistema, donde otro conmutador recibe los datos y los reenvía a los centros de procesamiento de cada uno de los cuatro sistemas remotos de detección de eventos.

Cada sistema recibe los datos, los procesa y genera una salida la cual a su vez constituye una entrada para el subsistema SPI. Dicha salida se refiere al tipo de evento detectado, con qué seguridad ha sido detectado y las coordenadas o localización del mismo. Además, cada sistema podrá enviar al subsistema SPI los parámetros recibidos desde los sensores con la finalidad que el subsistema SPI pueda verificar la certeza de su conclusión, corroborando dicha información con la proveniente de los restantes sistemas. De esta forma, el subsistema SPI recibe conclusiones y parámetros de todos los otros sistemas, analizando las conclusiones de cada sistema, los parámetros del ducto y las coordenadas del evento detectado. En base a toda esta información, el subsistema SPI genera una conclusión integral la cual envía al Sistema de Control y Monitoreo a Nivel Superior, el cual muestra en la pantalla, a través de un mapa y otros tipos de visualización, el evento, su tipo y las coordenadas calculadas. Con esta información, el operador de dicho sistema toma una decisión.

Como se puede apreciar en la Figura 3.10, el subsistema SPI poseerá una visión mucho más amplia y rica de los eventos generados por fugas y tomas clandestinas que la que pueda tener de forma individual cada uno de los sistemas remotos de detección de fugas, que efectúa el primer diagnóstico de detección de este tipo de eventos. Lo anterior se debe precisamente a la integración que hará el subsistema SPI de las salidas producidas por cada uno de dichos sistemas, lo cual sin lugar a dudas permitirá una mayor efectividad en la detección del evento.

La integración de las salidas producidas por los sistemas remotos de detección de fugas en un único patrón de entrada proporcionará al subsistema SPI un escenario muy rico para complementar, confirmar, discernir o descartar la detección de un evento particular reportado por uno o más de estos sistemas. Lo anterior es posible al contar con métodos de detección de fugas y tomas clandestinas que son por naturaleza complementarios, por ejemplo, al combinar información proveniente de

balance de masa con acústica, o balance de masa con fibra óptica. La integración de datos en el subsistema SPI además de incrementar la efectividad en la detección de eventos, también proporcionará un soporte de información muy valioso para la prevención de dichos eventos.



**Figura 3.10** Representación del flujo de datos y procesamiento en el subsistema SPI.

Como se puede apreciar en la Figura 3.10, el proceso de diagnóstico integral de detección de fugas y tomas clandestinas que ejecutará el subsistema SPI, se apoyará sobre el diagnóstico individual emitido por cada uno de los cuatro módulos de diagnóstico que componen el sistema. Como se describe en la Tabla 3.4, cada uno de estos módulos de diagnóstico se basa en una técnica de inteligencia artificial diferente, cada una de las cuales ha resultado ser un enfoque muy apropiado y robusto para contender con el problema de la detección de fugas y tomas clandestinas en ductos que transportan hidrocarburos.

**Tabla 3.4.** Los módulos de diagnóstico que integran el sistema PI.

Módulo de Diagnóstico	Técnica de inteligencia artificial en la que se basa
Módulo SE	Módulo de diagnóstico basado en un sistema experto con reglas de producción y representación de la incertidumbre acerca de los hechos que relaciona.
Módulo ARN1	Módulo de diagnóstico basado en una red neuronal artificial con arquitectura <i>perceptron</i> multiestrato y algoritmo de aprendizaje de retropropagación de errores.
Módulo ARN2	Módulo de diagnóstico basado en una red neuronal artificial recurrente con arquitectura <i>perceptron</i> multiestrato y algoritmo de aprendizaje de retropropagación de errores recurrente.
Módulo SLD	Módulo de diagnóstico basado en un sistema de lógica difusa.
Módulo DI	Módulo de diagnóstico integral el cual emula un sistema de votación y se basa sobre un pequeño conjunto de reglas lógicas. El módulo DI recibe como entrada un vector compuesto por las salidas producidas por los módulos SE, ARN1, ARN2, y SLD y produce un único valor de salida, correspondiente a la certeza de la detección de la fuga o toma clandestina.

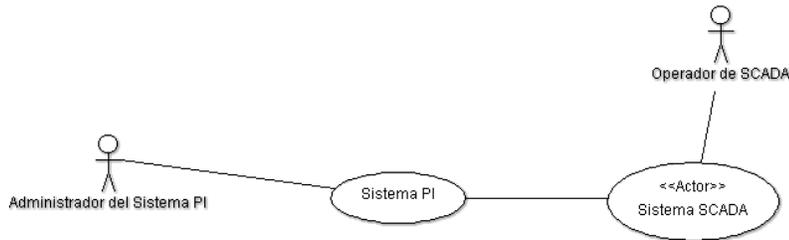
#### **Especificación de requerimientos del subsistema SPI: algunas de las técnicas utilizadas**

La Tabla 3.5 relaciona los requerimientos funcionales a nivel superior del subsistema SPI, como parte de la especificación complementaria. Las Figuras. 3.11 y 3.12 muestran dos de los diagramas de casos de uso desarrollados.

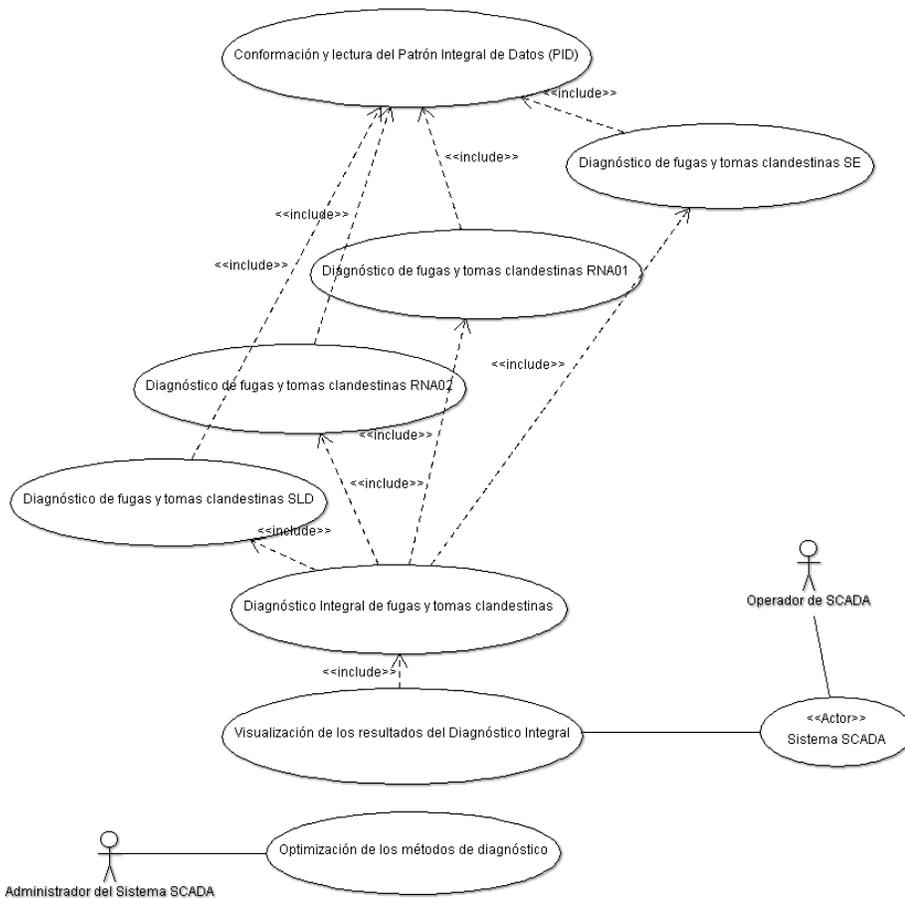
**Tabla 3.5.** Especificación complementaria. Los requerimientos funcionales.

Requerimiento funcional	Breve descripción
Conformación y lectura del patrón integral de datos	Integración en un patrón integral de datos (PID) de los datos de salida proporcionados por los sistemas remotos de detección de fugas. Presentación del PID a cada uno de los métodos de diagnóstico (SE, RNA01, RNA02 y SLD)
Diagnóstico de fugas y tomas clandestinas SE	Sistema Experto basado en Reglas de Producción con representación de la incertidumbre para el diagnóstico de fugas y tomas clandestinas.
Diagnóstico de fugas y tomas clandestinas RNA01	Red Neuronal Artificial " <i>backpropagation fully connected</i> " para el diagnóstico de fugas y tomas clandestinas.
Diagnóstico de fugas y tomas clandestinas RNA02	Red Neuronal Artificial recurrente para el diagnóstico de fugas y tomas clandestinas.
Diagnóstico de fugas y tomas clandestinas SLD	Sistema de Lógica Difusa para el diagnóstico de fugas y tomas clandestinas.

Diagnóstico integral de fugas y tomas clandestinas	Sistema de diagnóstico integral que basa su decisión final en los diagnósticos efectuados por los módulos SE, RNA01, RNA02 y SLD.
Optimización de los métodos de diagnóstico	Conjunto de técnicas de optimización que ajusta y maximiza el desempeño de los cuatro métodos de diagnóstico (SE, RNA01, RNA02 y SLD).
Visualización de los resultados del diagnóstico integral	Presentación en el sistema de visualización de SCADA de los resultados del diagnóstico integral.



**Figura 3.11** Diagrama de casos de uso a nivel contextual del subsistema SPI.



**Figura 3.12** Diagrama de casos de uso a nivel contextual del sistema SPI.

La Tabla 3.6 describe parte del glosario, relacionando las principales entidades a nivel superior identificadas. Finalmente, la Figura 3.13 muestra la primera versión del modelo del dominio del subsistema SPI.

Tabla 3.6. Glosario de las entidades a nivel superior del subsistema SPI.

Entidad	Alias o sigla	Breve descripción
Componente de Diagnóstico y Análisis SE	ComponenteSE	Su funcionalidad consiste en el diagnóstico de fugas y tomas clandestinas basado en la técnica de sistemas expertos. Recibe como entrada el patrón integral de datos (PID) y produce como salida la entidad DiagnosticoSE.
Componente de Diagnóstico y Análisis RNA01	ComponenteRNA01	Su funcionalidad consiste en el diagnóstico de fugas y tomas clandestinas basado en la técnica de red neuronal multiestrato con retropropagación de errores. Recibe como entrada el patrón integral de datos (PID) y produce como salida la entidad DiagnosticoRNA01.
Componente de Diagnóstico y Análisis RNA02	ComponenteRNA02	Su funcionalidad consiste en el diagnóstico de fugas y tomas clandestinas basado en la técnica de red neuronal multiestrato con retropropagación de errores recurrente. Recibe como entrada el patrón integral de datos (PID) y produce como salida la entidad DiagnosticoRNA02.
Componente de Diagnóstico y Análisis SLD	ComponenteSLD	Su funcionalidad consiste en el diagnóstico de fugas y tomas clandestinas basado en la técnica de lógica difusa. Recibe como entrada el patrón integral de datos (PID) y produce como salida la entidad DiagnosticoSLD.
Componente de Diagnóstico Integral	ComponenteMDI	Módulo cuya funcionalidad consiste en el diagnóstico integral final de fugas y tomas clandestinas. Se estructura como un conjunto de reglas lógicas. Recibe como entrada las entidades DiagnosticoSE, DiagnosticoRNA01, DiagnosticoRNA02 y DiagnosticoSLD, producidas como salida por los componentes de diagnóstico ComponenteSE, ComponenteRNA01, ComponenteRNA02 y ComponenteSLD, respectivamente.
Patrón Integral de Datos	PDI, Patrón PDI	Patrón integral de datos producido por el componente de control de datos (ComponenteCD). Se estructura a partir de información compatible en tiempo y localización de fugas y tomas clandestinas producida por los subsistemas remotos de detección de fugas.
DatoLC	SalidaLC	Entidad que representa la estructura de datos de salida de uno de los subsistemas basados en balance de masas. Los atributos de dicha estructura corresponden a información relativa a la fuga o toma clandestina generada por dicho subsistema, tales como diagnóstico, ubicación, tiempo, magnitud de la fuga, etc.
DatoVV	SalidaVV	Entidad que representa la estructura de datos de salida del otro subsistema basado en balance de masa. Los atributos de dicha estructura

		corresponden a información relativa a la fuga o toma clandestina generada por dicho subsistema, tales como diagnóstico, ubicación, tiempo, magnitud de la fuga, etc.
DatoAC	SalidaAC	Entidad que representa la estructura de datos de salida del subsistema basado en acústica. Los atributos de dicha estructura corresponden a información relativa a la fuga o toma clandestina generada por dicho subsistema, tales como diagnóstico, ubicación, tiempo, vibración, ruido, etc.
DatoFO	SalidaFO	Entidad que representa la estructura de datos de salida del subsistema basado en fibra óptica. Los atributos de dicha estructura corresponden a información relativa a la fuga o toma clandestina generada por dicho subsistema, tales como diagnóstico, ubicación, tiempo, vibración, ruido, deformación, etc.
Diagnóstico efectuado por el Componente de Diagnóstico y Análisis SE	DiagnosticoSE	Entidad que describe el resultado del diagnóstico de fugas o tomas clandestinas efectuado por el componente de diagnóstico SE.
Diagnóstico efectuado por el Componente de Diagnóstico y Análisis RNA01	DiagnósticoRNA01	Entidad que describe el resultado del diagnóstico de fugas o tomas clandestinas efectuado por el componente de diagnóstico RNA01.
Diagnóstico efectuado por el Componente de Diagnóstico y Análisis RNA02	DiagnósticoRNA02	Entidad que describe el resultado del diagnóstico de fugas o tomas clandestinas efectuado por el componente de diagnóstico RNA02.
Diagnóstico efectuado por el Componente de Diagnóstico y Análisis SLD	DiagnósticoSLD	Entidad que describe el resultado del diagnóstico de fugas o tomas clandestinas efectuado por el componente de diagnóstico SLD.
Diagnóstico efectuado por el Componente de Diagnóstico Integral	DiagnósticoMDI	Entidad que describe el resultado del diagnóstico final de fugas o tomas clandestinas efectuado por el componente de diagnóstico MDI.
Vector de datos LC del PID	DatoLC	Entidad que representa la estructura de datos de salida de uno de los subsistemas basado en balance de masa. Los atributos de dicha estructura corresponden a información relativa a la fuga o toma clandestina generada por dicho subsistema, tales como diagnóstico, ubicación, tiempo, magnitud de la fuga, etc.
Vector de datos VV del PID	DatoVV	Entidad que representa la estructura de datos de salida del otro subsistema basado en balance de masa. Los atributos de dicha estructura corresponden a información relativa a la fuga o

		toma clandestina generada por dicho subsistema, tales como diagnóstico, ubicación, tiempo, magnitud de la fuga, etc.
Vector de datos AC del PID	DatoAC	Entidad que representa la estructura de datos de salida del subsistema basado en acústica. Los atributos de dicha estructura corresponden a información relativa a la fuga o toma clandestina generada por dicho subsistema, tales como diagnóstico, ubicación, tiempo, vibración, ruido, etc.
Vector de datos FO del PID	DatoFO	Entidad que representa la estructura de datos de salida del subsistema basado en fibra óptica. Los atributos de dicha estructura corresponden a información relativa a la fuga o toma clandestina generada por dicho subsistema, tales como diagnóstico, ubicación, tiempo, vibración, ruido, deformación, etc.

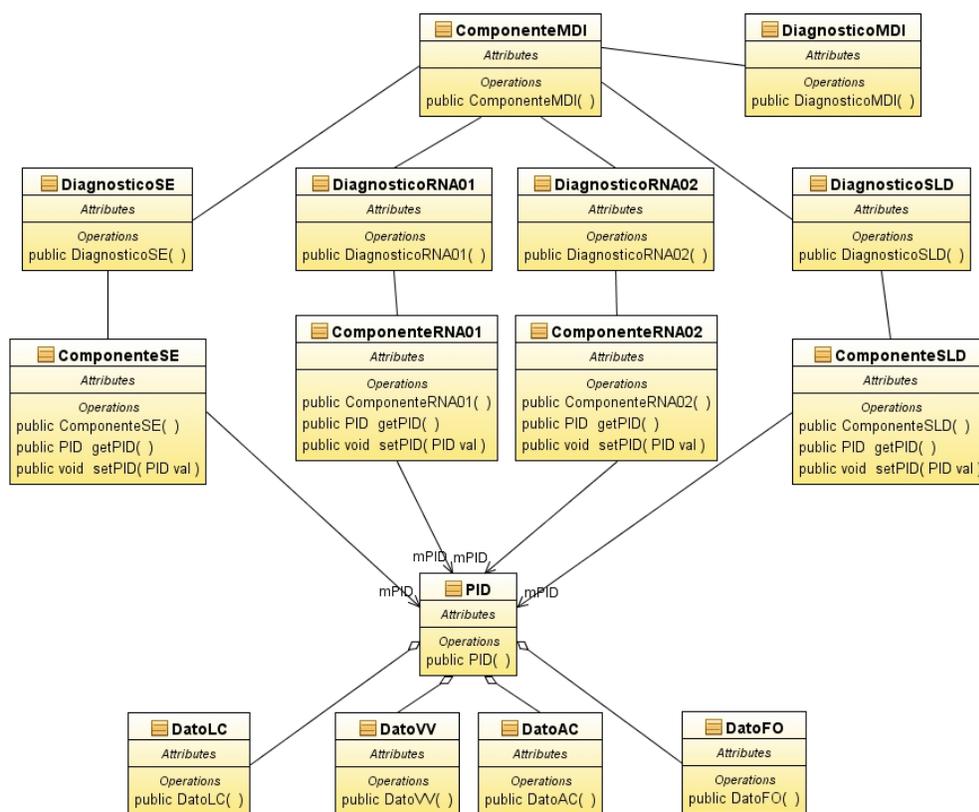


Figura 3.13 Modelo del dominio del sistema SPI.

### III.3.2 La plataforma bioinformática Evolution

#### ***Declaración del alcance de la plataforma bioinformática Evolution: objetivos, visión y alcance***

La plataforma bioinformática Evolution es una herramienta computacional dedicada al modelado y simulación del plegamiento de secuencias de aminoácidos [16]. Evolution integra la destreza de los algoritmos evolutivos y otras técnicas computacionales en una arquitectura multiagente. Evolution permite al usuario (biólogo, bioquímico, biofísico, bioinformático, etc.) estudiar y explorar problemas complejos de la biología estructural, tales como el plegamiento de proteínas, proporcionando los medios para experimentar con el sistema, así como para desarrollar nuevas funcionalidades y algoritmos para adaptar la infraestructura a las características y singularidades del problema a abordar.

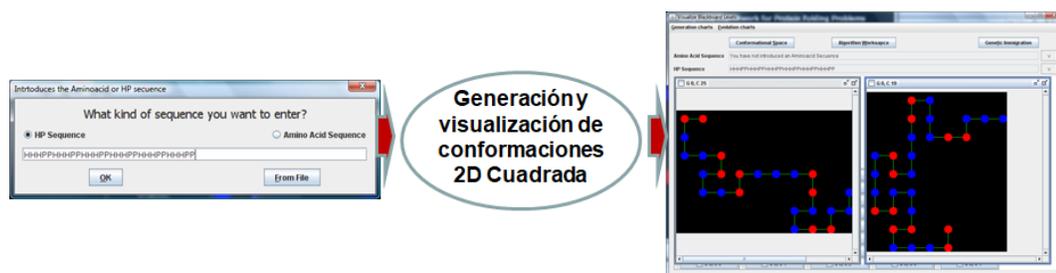
Evolution también permite al usuario realizar un seguimiento no sólo de los resultados obtenidos para el problema biológico bajo estudio y exploración, sino también de la eficiencia de los algoritmos, la adecuación de la función de fitness y de otros parámetros del proceso de optimización.

Entre las principales prestaciones que proporciona la plataforma Evolution al usuario, se encuentran las siguientes:

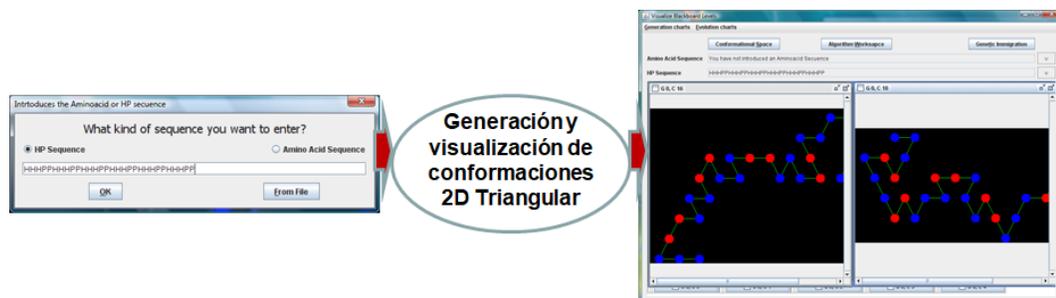
1. Generación y visualización de una población aleatoria de conformaciones 2D Cuadrada a partir de una secuencia HP proporcionada como entrada.
2. Interacción con las conformaciones 2D Triangular.
3. Generación y visualización de una población aleatoria de conformaciones 2D Triangular a partir de una secuencia HP proporcionada como entrada.
4. Interacción con las conformaciones 2D Triangular.
5. Generación y visualización de una población aleatoria de conformaciones 3D Cúbica a partir de una secuencia HP proporcionada como entrada.
6. Interacción con las conformaciones 3D Cúbica.
7. Generación y visualización de los gráficos 2D (fitness vs. conformación, radio de giro vs. Conformación, etc.) para describir las características del espacio de conformaciones generado.
8. Optimización de la población de conformaciones generada a través del uso de un algoritmo genético que integre una amplia gama de operadores genéticos y técnicas genéticas.
9. Generación y visualización de los gráficos 2D (fitness vs. generación, radio de giro promedio vs. generación, distancia máxima vs. generación, relación fitness padres/hijo, etc.) para describir las características del proceso de optimización a través de las diferentes generaciones creadas.
10. Mecanismo de migración genética que permita retomar una generación particular creada por el algoritmo genético y sustituir algunas de sus conformaciones por nuevo material genético creado de forma aleatoria.

**Especificación de requerimientos de la plataforma bioinformática Evolution: algunas de las técnicas utilizadas**

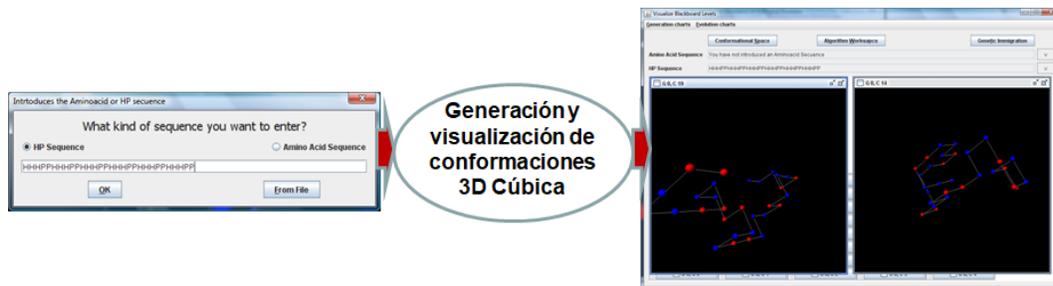
Las Figuras 3.14 a la 3.20 proporcionan una especificación gráfica de los principales requerimientos de la plataforma bioinformática *Evolution*, lo cual facilita la comprensión de los mismos.



**Figura 3.14** Especificación gráfica de los requerimientos funcionales de la plataforma bioinformática Evolution. Generación y visualización de conformaciones 2D Cuadrada.

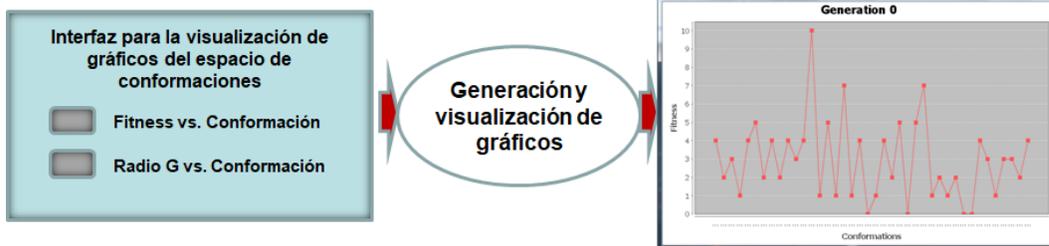


**Figura 3.15** Especificación gráfica de los requerimientos funcionales de la plataforma bioinformática Evolution. Generación y visualización de conformaciones 2D Triangular.

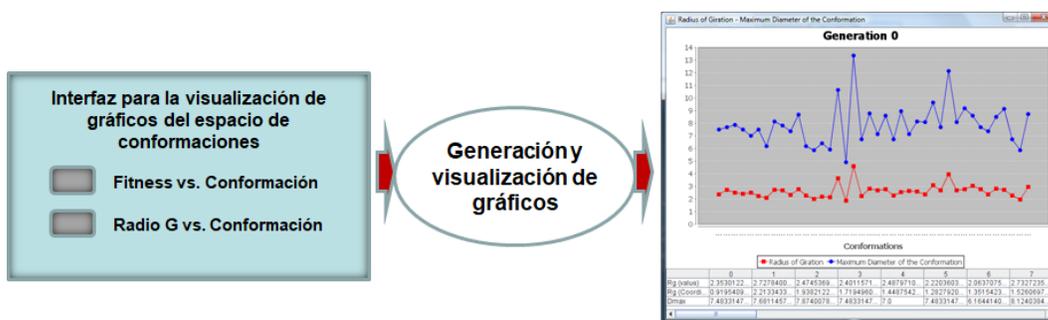


**Figura 3.16** Especificación gráfica de los requerimientos funcionales de la plataforma bioinformática Evolution. Generación y visualización de conformaciones 3D Cúbica.

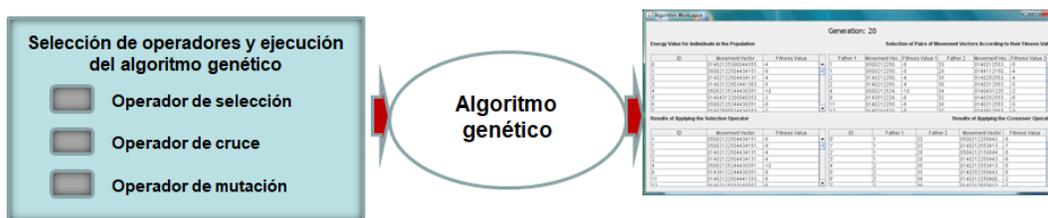
El proceso de Requerimientos  
— Casos de estudio



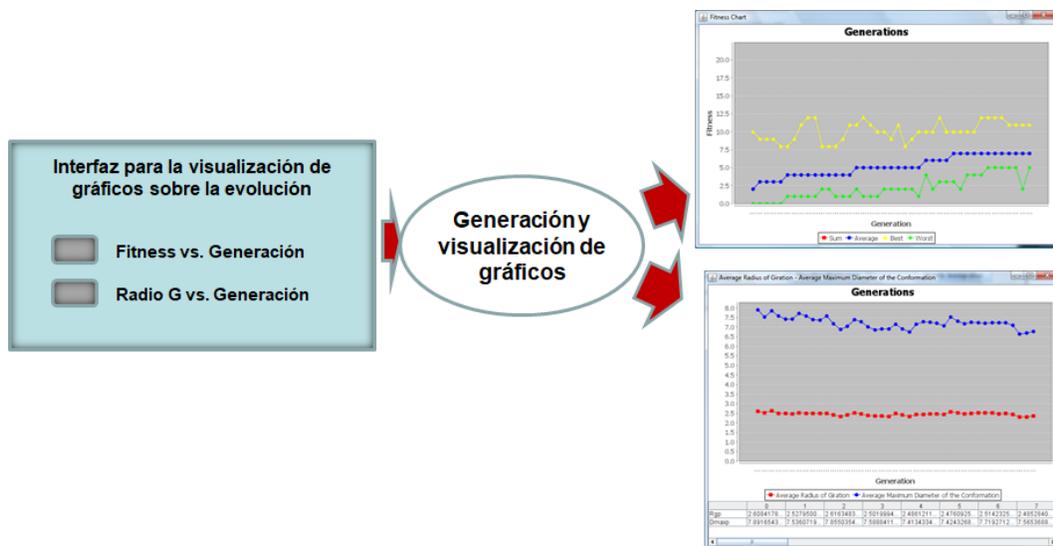
**Figura 3.17** Especificación gráfica de los requerimientos funcionales de la plataforma bioinformática Evolution. Generación y visualización de gráficos.



**Figura 3.18** Especificación gráfica de los requerimientos funcionales de la plataforma bioinformática Evolution. Generación y visualización de gráficos.

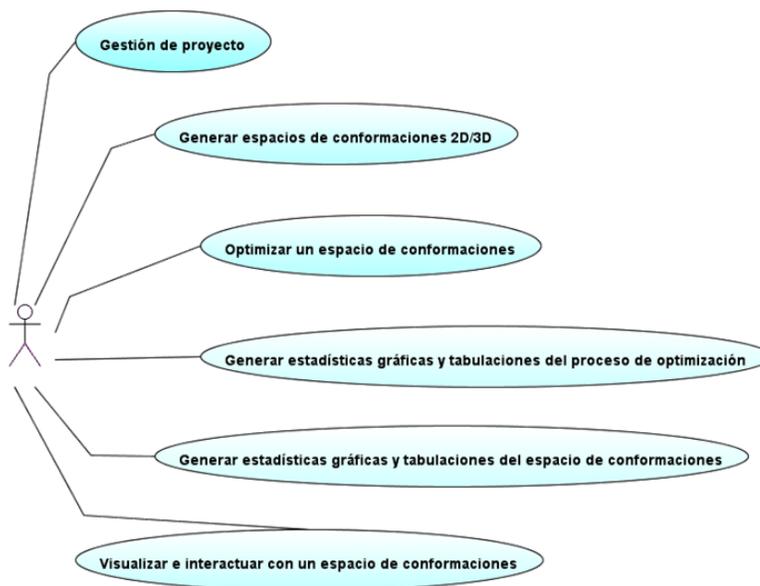


**Figura 3.19** Especificación gráfica de los requerimientos funcionales de la plataforma bioinformática Evolution. Generación y visualización de tablas.



**Figura 3.20** Especificación gráfica de los requerimientos funcionales de la plataforma bioinformática Evolution. Generación y visualización de gráficos.

La Figura 3.21 ilustra un diagrama de casos de uso a nivel superior, el cual refleja los principales tipos de prestaciones que la plataforma bioinformática *Evolution* pone a disposición del usuario. Por otra parte, las Tablas 3.7 y 3.8 relacionan parte del glosario y de la especificación complementaria, respectivamente. Finalmente, la Figura 3.22 muestra el primer modelo del dominio de la plataforma bioinformática *Evolution*.



**Figura 3.21** Diagrama de casos de uso a nivel superior. Principales tipos de prestaciones que la plataforma bioinformática Evolution pone a disposición del usuario.

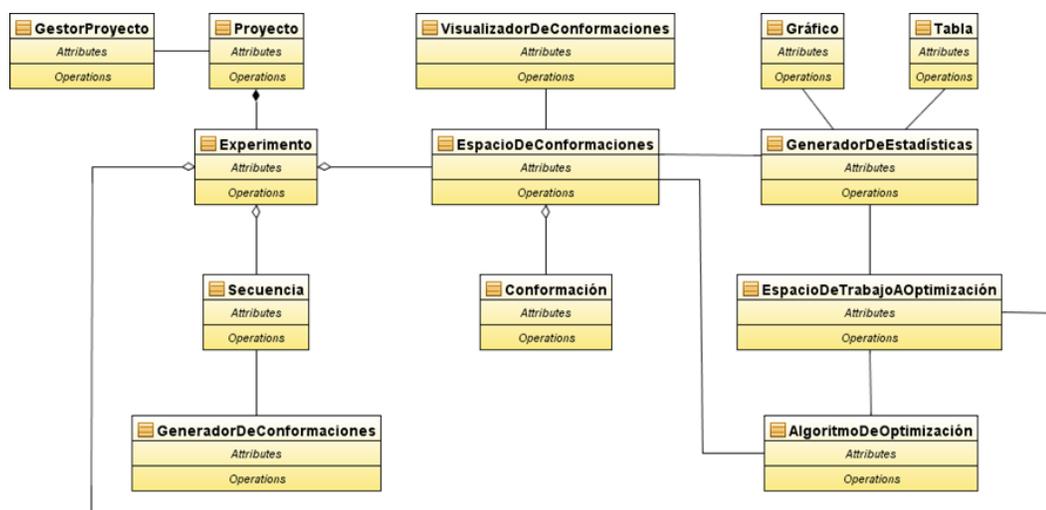
**Tabla 3.7.** Glosario de las entidades a nivel superior de la plataforma bioinformática *Evolution*. Fragmento.

Entidad	Breve descripción
<b>Secuencia</b>	Cadena alfabética de longitud $l$ , escrita a partir de un alfabeto de 2 caracteres (H, P) o de un alfabeto de 20 caracteres (los 20 aminoácidos que conforman a las proteínas).
<b>Conformación</b>	Conjunto de coordenadas (puntos) obtenidos a partir de una <i>secuencia</i> . Se obtiene ejecutando de forma aleatoria los movimientos admitidos en una rejilla 2D (hacia adelante, a la derecha, a la izquierda) o en una rejilla 3D (hacia adelante, a la derecha, a la izquierda, hacia arriba, hacia abajo, hacia atrás). Los movimientos se ejecutan tomando como vector de referencia la línea que une las dos primeras cuentas (aminoácidos), las cuales se consideran fijas en la rejilla.
<b>Espacio de Conformaciones</b>	Espacio de trabajo donde se generan y visualizan las conformaciones 2D/3D, a partir de una secuencia HP. El espacio de conformaciones permite la sección de conformaciones particulares y la interacción con las mismas.
<b>Generador de Conformaciones</b>	Entidad que crea un número fijo de conformaciones aleatorias a partir de una <i>secuencia</i> HP recibida como entrada. El generador de conformaciones produce conformaciones 2D Cuadrada, 2D Triangular y 3D Cúbica. Dada su complejidad, esta entidad “abstracta” envuelve un conjunto de otras entidades que interactúan entre sí para generar el tipo y cantidad de conformaciones requeridas.
<b>Generador de Estadísticas</b>	Entidad encargada de producir la información estadística en forma de gráficos y tablas, proporcionando de esta forma al usuario otra herramienta más para la interpretación de los resultados parciales y finales del proceso de optimización. En particular, se requieren gráficos del tipo Fitness Máximo/Fitness Mínimo/Fitness Promedio vs. Generación, Valor de Fitness vs. Conformación, así como tablas que muestren las relaciones entre secuencias padres y secuencias descendientes, etc.
<b>Grafo Estadístico</b>	Entidad de tipo gráfico producida por la entidad Generador de Estadísticas. Un grafo estadístico muestra diferentes aspectos de un espacio de conformaciones, ya sea a través de curvas que describen el comportamiento del proceso de optimización en el tiempo, como a través de la visualización de la geometría de las conformaciones generadas.
<b>Tabla Estadística</b>	Entidad de tipo tabla, producida por la entidad Generador de Estadísticas. La herramienta computacional permite la generación y visualización de diferentes tipos de tabla, en dependencia de los criterios introducidos por el usuario.

<p><b>Algoritmo de Optimización</b></p>	<p>Algoritmo evolutivo para hacer evolucionar a través de un proceso de optimización un espacio de conformaciones generado de forma aleatoria. Esta entidad está compuesta por otros componentes que modelan los diferentes operadores genéticos y técnicas genéticas comúnmente utilizados. Ejemplos de estos operadores y técnicas son los siguientes: operador de selección, operador de cruce, operador de mutación, técnica de elitismo, etc.</p>
---	--

**Tabla 3.8.** Especificación complementaria desarrollada durante el análisis de requerimientos de la plataforma bioinformática Evolution. Fragmento.

Especificación complementaria
<p>Existen varios tipos de conformaciones, en dependencia del <i>lattice</i> (rejilla) donde vienen creadas. En este sentido, el sistema deberá crear, visualizar, manipular y optimizar tres tipos diferentes de espacios de conformaciones: 2D Cuadrada, 2D Triangular y 3D Cúbica.</p>
<p>El algoritmo evolutivo, a través del cual se lleva a cabo el proceso de optimización, está compuesto por varios operadores y técnicas genéticos. Los operadores a diseñar e implementar son específicamente el operador de selección “Ruleta Pesada”, Selección por “Torneo”, Selección “<i>Top Percent</i>” y Selección “<i>Population Decimation</i>”, el operador de Cruce en un Punto, en dos Puntos y Cruce Uniforme, y los operadores de Mutación Simple y Aleatoria los cuales son invocados por el operador de Cruce. Por otra parte, la técnica genética a implementar será el Elitismo, el cual permitirá mantener la mejor conformación (o las mejores conformaciones) de una generación <math>k</math> a una generación <math>k+1</math>.</p>
<p>Los operadores y técnicas genéticos estarán disponibles de acuerdo al tipo de algoritmo genético seleccionado. La plataforma bioinformática se basa en el trabajo de dos tipos de algoritmo genético: el Algoritmo Simple y el Algoritmo RankGA [15].</p>



**Figura 3.22** Modelo del Dominio inicial de la plataforma bioinformática Evolution.

### III.4 Referencias del capítulo

1. Cusumano, M.A. The Factory Approach to Large-Scale Software Development: Implications for Strategy, Technology, and Structure. Classic Reprints Series, 2017.
2. Garland, J., Anthony, R. Large Scale Software Architecture. A Practical Guide Using UML. Wiley, 2003.
3. IEEE-STD-830-1998: Recommended Practice for Software Requirements Specifications.
4. Janakiram, D. Building Large Scale Software Systems. McGraw Hill Education, 2013.
5. Lakos, J. Large-Scale C++ Software Design. Addison Wesley, 1996.
6. Screerer, A. Coordination in Large-Scale Agile Software Development: Integrating Conditions and Configurations in Multiteam Systems (Progress in IS). Springer, 2017.
7. Gómez Fuentes, M.C. Notas del Curso Análisis de Requerimientos. Universidad Autónoma Metropolitana. 2011.
8. Gómez Fuentes, M.C., Cervantes Ojeda, J., González-Pérez, P.P. Fundamentos de Ingeniería de Software. Universidad Autónoma Metropolitana, 2019.
9. Pfleeger, S. L. Ingeniería de software: Teoría y práctica. Pearson Education, 2002.
10. Pressman, R. S. Ingeniería del software: Un enfoque práctico. McGraw-Hill, 2010.
11. Sommerville, I. Ingeniería del software. Pearson Addison Wesley, 2012.
- Black, R. Managing the testing process. Wiley, 2009.
12. Burnstein, I. Practical software testing: a process-oriented approach. Springer, 2013.
13. Tsui, F., Karam, O., Bernal, B. Essentials of software engineering. Jonas & Bartlett.
14. González-Pérez, P.P., Sadovnychyy, A., Alarcón Ramos, L.A., González González, D., Elorza Gómez, K., Peña Falcón, J.L. Sistema para la detección de fugas y tomas clandestinas en ductos de transporte de gas y líquidos. Sistema Computacional de Procesamiento Integral de Datos (PI). Documento de Análisis y Diseño. Reporte Técnico. 2013.
15. González-Pérez, P.P., Schaum, A., Sadovnychyy, A., Bernal Jaquez, R., Méndez Gurrola, I.I., Alarcón Ramos, L.A., Rosales Cruz, L. Sistema para la detección de fugas y tomas clandestinas en ductos de transporte de gas y líquidos. Sistema Computacional de Procesamiento Integral de Datos (PI). Estado del Arte. Reporte Técnico. 2011.
16. Sánchez Gutiérrez, M.E. Plataforma Bioinformática para el Estudio, Modelado y Simulación del Plegamiento de Proteínas. Proyecto Terminal. Universidad Autónoma Metropolitana, Unidad Cuajimalpa. 2010.

# Capítulo IV Gestión de los Requerimientos No Funcionales

## IV.1 Los requerimientos no funcionales

En el desarrollo de software, los requerimientos no funcionales (o restricciones) se refieren a aquellas exigencias, necesidades o restricciones que debe satisfacer el sistema de software una vez concluido y liberado, pero que no forman parte de las funcionalidades o prestaciones específicas que prestará el sistema durante su ejecución. En otras palabras, los requerimientos no funcionales, a diferencia de los requerimientos funcionales, describen características del funcionamiento del sistema de software o de éste como entidad.

Como ya mencionamos, una de las principales características de los sistemas de software a gran escala es la dependencia de una gran variedad de requerimientos no funcionales [1-5], clasificados en dos principales categorías: a) los requerimientos no funcionales que caracterizan el funcionamiento global del sistema software y b) los requerimientos no funcionales que caracterizan al sistema software como producto.

Los requerimientos no funcionales constituyen un aspecto clave con el que se debe contender en el desarrollo de un sistema de software a gran escala. De aquí la imperiosa necesidad de identificar, especificar y analizar los requerimientos no funcionales del sistema de software durante el “Proceso de requerimientos”, a los cuales se les debe dar tanta importancia como a los requerimientos funcionales.

Por ejemplo, si pensamos en el desarrollo de un sistema Web del tipo Banca en Línea, no tendremos dificultad para identificar como requerimientos funcionales “consulta de saldo”, “pago de servicios” y “transferencia a otros bancos”; mientras que requerimientos tales como “seguridad”, “portabilidad” y “robustez” caen en la categoría de no funcionales, ya que éstos no se refieren a funcionalidades específicas del sistema Web Portal Bancario, sino a características que la ejecución del sistema o el sistema per sé debe exhibir.

Esta primera caracterización proporcionada de los requerimientos no funcionales podemos enriquecerla con el siguiente principio: a diferencia de los requerimientos funcionales, los cuales dependen y son determinados por el alcance y dominio del

sistema de software a desarrollar; los requerimientos no funcionales no dependen del alcance y dominio de la aplicación, éstos representan características necesarias que debe exhibir el sistema de software durante su ejecución y/o como entidad o producto final.

De aquí que diferentes sistemas de software, concluidos o en desarrollo, con requerimientos funcionales muy disímiles, puedan compartir prácticamente los mismos requerimientos no funcionales. Para ilustrar lo anterior, consideremos los siguientes tres ejemplos de sistemas de software desarrollados y en ejecución:

- 1) Sistema Web Portal Bancario.
- 2) Sistema Web de Reservaciones y Ventas de Billetes de una Aerolínea.
- 3) Sistema de Control y Monitoreo de Fugas en Ductos que Transportan Hidrocarburos.

A partir del propio nombre de cada uno de los sistemas de software antes relacionados, podemos comprender que el conjunto de requerimientos funcionales de cada uno de éstos será muy específico y particular del dominio o alcance de la aplicación. Por ejemplo, mientras que el Sistema Web Portal Bancario exhibirá funcionalidades tales como “consulta de saldo”, “pago de servicios” y “transferencia a otros bancos”; el Sistema Web de Reservaciones y Ventas de Billetes de una Aerolínea proporcionará funcionalidades tales como “selección de vuelos”, “introducción de datos de los pasajeros” y “reservación y pago de vuelos”; y finalmente, el Sistema de Control y Monitoreo de Fugas en Ductos que Transportan Hidrocarburos incluirá funcionalidades tales como “cálculo de balance de masa”, “monitoreo de ruidos y vibraciones en el ducto” y “activación de alarma”. Sin embargo, seguramente los tres sistemas compartirán requerimientos no funcionales tales como “confiabilidad”, “robustez”, “seguridad”, etc.

Hasta ahora nos hemos referido a los requerimientos no funcionales intentando separarlos en dos principales categorías:

- Requerimientos no funcionales que caracterizan la ejecución o funcionamiento global del sistema de software.
- Requerimientos no funcionales que caracterizan al sistema de software como producto, comúnmente relacionados con el ulterior proceso de mantenimiento, mejoras o crecimiento funcional del mismo.

Con esta idea en mente, podemos identificar los principales requerimientos no funcionales en cada una de estas categorías y referirnos a la gestión de los mismos.

### IV.1.1 Gestión de los requerimientos no funcionales que caracterizan la ejecución o funcionamiento global del sistema de software

Entre los principales requerimientos no funcionales relacionados con el funcionamiento o ejecución del sistema software se pueden identificar los siguientes:

- *Confiabilidad*: Debe existir una baja probabilidad de que el sistema presente fallas.
- *Robustez*: El sistema debe responder adecuadamente cuando se presenten situaciones fuera de lo común.
- *Seguridad*: Que la información en el sistema esté protegida contra atacantes que pudieran causar daño.
- *Portabilidad*: Que el sistema se pueda instalar con relativa facilidad en diferentes ambientes de hardware.
- *Eficiencia o Rendimiento*: Que se consiga hacer la mayor cantidad de trabajo (calcular resultados, enviar mensajes, desplegar interfaces de usuario, etc.) con un uso mínimo de recursos (memoria, tiempo de procesamiento, ancho de banda, etc.).

Nótese que en cada uno de los casos anteriores es necesario definir una métrica que permita cuantificar el requerimiento mínimo (o máximo en su caso).

### IV.1.2 Gestión de los requerimientos no funcionales que caracterizan al sistema de software como producto

Ejemplos de requerimientos no funcionales relacionados con el sistema de software como entidad o producto:

- *Reusabilidad*: Que el sistema o algunas de sus partes se puedan utilizar en la construcción de otros sistemas.
- *Flexibilidad*: Que el sistema se pueda adaptar a otras circunstancias agregando o modificando módulos.
- *Escalabilidad*: Se refiere a la capacidad del sistema de software de aumentar su rendimiento en la medida en que nuevos recursos de hardware son incorporados.
- *Facilidad de pruebas*: Se refiere al nivel de complejidad de las pruebas efectuadas para asegurar la calidad del sistema de software.

## IV.2 Caso de estudio

### IV.2.1 Subsistema SPI, del sistema de detección de fugas en ductos que transportan hidrocarburos. gestión de los requerimientos no funcionales

Como ya nos referimos en el Epígrafe 1.6 (ver Tabla 1.2), una de las características que hacen del subsistema SPI un sistema de software a gran escala, es precisamente la dependencia de una gran variedad de requerimientos no

funcionales. Este hecho impacta significativamente en la complejidad del proceso de desarrollo del subsistema SPI. La Tabla 4.1 relaciona dichos requerimientos no funcionales e indica una propuesta inicial de gestión para cada uno de éstos.

**Tabla 4.1.** Gestión de los requerimientos no funcionales en el sistema PI.

Tipo de restricción	Descripción / Propuesta inicial de gestión
Arquitectura lógica colaborativa y arquitectura física distribuida	<p>Dada la naturaleza de trabajo colaborativo que caracteriza al sistema SPI, la arquitectura lógica del mismo debe garantizar el trabajo colaborativo, oportunista e incremental de los diferentes componentes que integran el sistema. De aquí que, un enfoque basado en la composición de la arquitectura de pizarra (<i>blackboard architecture</i>) y los sistemas orientados a eventos, sea la opción más adecuada para la arquitectura lógica. En cuanto a la arquitectura física, ésta necesariamente tendrá que ser una arquitectura distribuida compuesta por al menos tres nodos físicos.</p>
Control del nivel de acceso	<p>En función del tipo de usuario que tendrá acceso al sistema SPI, se podrá detallar a que módulos podrá acceder o que acciones podría en un determinado momento efectuar. Para el sistema SPI deben definirse los niveles de acceso que correspondan a los siguientes usuarios o actores:</p> <ul style="list-style-type: none"> <li>▪ <b>Operador del Sistema de Control y Monitoreo a Nivel Superior</b> – usuario pasivo, solo recibe información visual.</li> <li>▪ <b>Administrador</b> - puede corresponder a dos modalidades: i) para atenderlas cuestiones relacionadas con el mantenimiento del sistema, y ii) para ejecutar cada vez que sea necesario los algoritmos de optimización de los métodos de diagnóstico.</li> </ul>
Interfaces de usuario e interfaces de comunicación con otros sistemas	<ul style="list-style-type: none"> <li>▪ <b>Interfaz con el sistema de Control y Monitoreo a Nivel Superior.</b> Es importante determinar de forma precisa de qué forma dicho nos podría proporcionar información sobre los ductos, básicamente como nos podríamos comunicar con sus datos; por ejemplo, para obtener temperatura o presión en un determinado momento. Por otra parte, el subsistema SPI utilizará el subsistema de visualización de dicho sistema para visualizar sus salidas.</li> <li>▪ <b>Interfaces con el usuario.</b> Aunque aún no se ha determinado con claridad cuál será el tipo de</li> </ul>

	<p>usuario del subsistema SPI, al menos ya se han acumulado muchos elementos que nos hacen pensar en dos tipos principales de usuario: el operador del Sistema de Control y Monitoreo a Nivel Superior, y el administrador del subsistema SPI, tal como se describió en la restricción “Nivel de acceso”. En base a lo anterior, será necesario definir las características de las interfaces para ambos tipos de usuario.</p> <ul style="list-style-type: none"> <li>▪ <b>Interfaces con otros subsistemas.</b> Cada uno de los subsistemas remotos de detección de fugas, que proporciona sus salidas al subsistema SPI, debe entregar los datos en un determinado formato, de tal manera que el sistema integral pueda procesar dicha información. En este caso, no solo debemos gestionar la forma en que debe proceder la comunicación, sino también el formato que deberán tener los datos.</li> </ul>
Seguridad	<p>La ejecución del subsistema SPI se realiza de forma independiente a la del Sistema de Control y Monitoreo a Nivel Superior, con la finalidad de no entorpecer su operación. Los datos que el subsistema SPI manipule estarán bajo el control del administrador del subsistema SPI, y los resultados que el subsistema SPI produzca sólo podrán ser vistos por la persona que el administrador designe, que en primera instancia sería el operador del Sistema de Control y Monitoreo a Nivel Superior.</p>
Robustez	<p>El subsistema SPI es un sistema de ejecución de tareas en tiempo real, el cual recibe en tiempo real datos de eventos de detección de fugas y tomas clandestinas desde cuatro subsistemas remotos. Se requiere que el subsistema SPI sea lo suficientemente robusto para ejecutar el diagnóstico integral, cuando al menos uno de estos sistemas de detección esté comunicando algún evento de detección de fuga. Es decir, el subsistema SPI debe ser capaz de contender con la incompletitud de los datos de entrada.</p>
Confiabilidad	<p>En función del número de subsistemas de entrada y de la confiabilidad de los datos de salida que éstos proporcionan estará determinada la confiabilidad del subsistema SPI. En cierto sentido, la confiabilidad de las salidas del sistema SPI dependerá de la confiabilidad de sus entradas. No obstante, el subsistema SPI debe proporcionar a través de valores de probabilidad o certidumbre la confiabilidad del diagnóstico integral producido como salida.</p>

<p>Sincronización de las entradas recibidas y sincronización de los datos recibidos en cuanto a tiempo y localización.</p>	<p>La sincronización de datos constituye una de las restricciones más fuertes a considerar durante el desarrollo del subsistema SPI. Será necesario considerar dos tipos de sincronización:</p> <p><b>Entradas al subsistema SPI.</b> ¿Con qué frecuencia los sistemas que alimentarán al subsistema SPI entregarán sus datos de salida a éste para que los mismos puedan ser procesados? De forma general, los sistemas de detección de fugas basados en balance de masa son un poco más lentos que los sistemas basados en acústica y fibra óptica, debido a que el procesamiento que ejecutan los primeros requiere más tiempo. El subsistema SPI estaría condicionado a procesar a la velocidad de los sistemas de detección de fugas más lentos para el diagnóstico, pero no así para la prevención, en donde la velocidad de respuesta dependería del tiempo de procesamiento de los sistemas que no se basan en balance de masas y que son más rápidos. El subsistema SPI puede trabajar con sincronización por eventos, lo cual le permitiría analizar datos de entrada inmediatamente, en tiempo real. Comúnmente, los sistemas de detección de fugas tienen diferente tiempo de respuesta, por lo que datos relativos al mismo evento llegarán al subsistema SPI en tiempos diferentes. Además, cada subsistema determina el evento con diferente precisión de localización, por lo tanto, una tarea adicional para el subsistema SPI será sincronizar los datos de cada subsistema por tiempo y por localización para no analizar datos de un mismo evento como datos de eventos diferentes.</p> <p><b>Salidas del subsistema SPI.</b> En este caso la velocidad con la que el subsistema SPI entregará los datos de prevención y diagnóstico al usuario dependerá de la velocidad con la que reciba los datos de entrada y del hardware en donde se ejecuta el mismo. El subsistema SPI requiere ofrecer al usuario respuesta en tiempo real.</p>
<p>Robustez respecto a la omisión de datos de entrada</p>	<p>Cada uno de los sistemas remotos de detección de fugas proporcionará sus datos de salida como entrada al subsistema SPI. Por lo tanto, la cantidad de subsistemas que se encuentren funcionando en un momento dado determinará la confiabilidad de la salida del sistema PI para ese momento.</p>
<p>Plataforma/Sistema Operativo</p>	<p>Debido a que la mayoría de los sistemas que se emplean en el entorno del sistema de control y monitoreo a nivel superior son servidores basados en procesadores Intel, con sistema operativo (SO) Windows Server, el subsistema SPI estaría enfocado a ejecutarse en este tipo de plataforma. Las características de la plataforma también definen la</p>

	<p>velocidad de procesamiento y, por lo tanto, la velocidad con la que el subsistema SPI proporcionará sus salidas. Para no afectar al sistema de control y monitoreo a nivel superior (ni afectar su funcionamiento), el subsistema SPI empleará una plataforma separada. En una primera etapa (prototipo) se podrá usar una máquina virtual para hacer pruebas y determinar recursos necesarios.</p>
--	--

### IV.3 Referencias del capítulo

1. Cusumano, M.A. The Factory Approach to Large-Scale Software Development: Implications for Strategy, Technology, and Structure. Classic Reprints Series, 2017.
2. Garland, J., Anthony, R. Large Scale Software Architecture. A Practical Guide Using UML. Wiley, 2003.
3. Janakiram, D. Building Large Scale Software Systems. McGraw Hill Education, 2013.
4. Lakos, J. Large-Scale C++ Software Design. Addison Wesley, 1996.
5. Screerer, A. Coordination in Large-Scale Agile Software Development: Integrating Conditions and Configurations in Multiteam Systems (Progress in IS). Springer, 2017.

# Capítulo V **Diseño Arquitectónico y Diseño de la Interacción entre Interfaces Gráficas de Usuario**

La fase de diseño en la construcción de sistemas de software es esencial para el éxito de un proyecto, ya que las decisiones más importantes de la construcción del sistema se toman en esta fase. El objetivo de un buen diseño es que se determinen los módulos con los que contará el sistema, las tareas que cada uno de estos deberá llevar a cabo y la manera que interactúan entre ellos. También se definen las estructuras de datos que se utilizarán, e incluso se puede especificar la base de la lógica para codificar de tal manera que se cumpla con la especificación de requerimientos. Se espera que la fase de codificación se simplifique siguiendo las decisiones que están documentadas en el documento de diseño.

El documento de diseño debe basarse en la Especificación de Requerimientos. Un buen diseño debe cumplir con todos los requerimientos asignando a cada módulo tareas específicas para cada servicio de tal manera que la implementación de cada módulo pueda ser realizada por diferentes programadores de forma independiente. En el Desarrollo de Software a Gran escala, es fundamental contar con un buen documento de diseño, ya que éste será la documentación que facilite las pruebas y el mantenimiento.

## **V.1 Arquitectura lógica del sistema de software**

Como ya se mencionó en el Capítulo I una de las principales características del software a gran escala (ver Tabla 1.1) es que generalmente requiere de la distribución del procesamiento y de la información en diferentes módulos lógicos que comúnmente serán asignados a diferentes nodos físicos. Por otra parte, y a la vez estrechamente relacionado con lo antes expuesto, la modularización y el fuerte desacoplamiento son otras de las características relevantes del desarrollo de software a gran escala que impactan en la arquitectura del mismo. Estos aspectos conllevan a la toma de decisiones tanto con relación a la arquitectura lógica como a la arquitectura física, que resulten más adecuadas para satisfacer los requerimientos y restricciones del sistema de software a gran escala en desarrollo.

La arquitectura lógica del sistema de software (comúnmente referida como arquitectura del software, o simplemente arquitectura lógica) especifica la organización del software, considerando los componentes que integrarán el sistema, las interfaces y comportamientos que caracterizan a estos componentes, así como la forma en que se establece la comunicación, interacción y colaboración entre los mismos. La arquitectura lógica integra todos los aspectos del software: lógicos, de proceso, de componentes, físicos, entre otros.

La arquitectura lógica abarca decisiones importantes acerca de:

- La organización del sistema de software.
- Los elementos estructurales que compondrán el sistema y sus interfaces, junto con sus comportamientos, tal y como se especifican en las colaboraciones entre estos elementos.
- La composición de los elementos estructurales y del comportamiento en subsistemas progresivamente más grandes.
- El estilo de la arquitectura que guía esta organización: los elementos y sus interfaces, sus colaboraciones y su composición.

La arquitectura lógica está afectada no sólo por la estructura y el comportamiento, sino también por el uso, la funcionalidad, el rendimiento, la flexibilidad, la reutilización, la facilidad de comprensión, las restricciones y compromisos económicos y tecnológicos, y la estética.

Se necesita una arquitectura lógica para:

- Comprender el sistema.
- Organizar el desarrollo.
- Fomentar la reutilización.
- Hacer evolucionar el sistema.

Dentro de los principales enfoques a la arquitectura lógica del software, los más comúnmente utilizados, generalizados y extendidos en las últimas décadas han sido los siguientes:

- Modelo arquitectónico orientado al flujo de datos.
- Modelo arquitectónico orientado a objetos.
- Modelo arquitectónico de sistemas de software interactivos.
- Modelo arquitectónico de pizarra o repositorio (del inglés, *blackboard systems*).

Debido al enfoque que se le pretende dar al presente material —es decir, su orientación al desarrollo de sistemas de software a gran escala— aquí solo nos

referiremos al modelo arquitectónico de sistemas de software interactivo, modelo-vista-controlador (MVC) [1-4] y al modelo arquitectónico de pizarra o repositorio [6-8], ambos ampliamente utilizados y difundidos en el desarrollo de software a gran escala en las últimas décadas. No obstante, vale la pena mencionar que tanto el modelo arquitectónico de sistemas de software interactivos MVC como el modelo arquitectónico de pizarra se erigen sobre el modelo arquitectónico orientado a objetos. De hecho, el modelo arquitectónico de sistemas de software interactivo nace en el contexto de los sistemas orientados a objetos. Por otra parte, aunque el modelo arquitectónico de pizarra surge de forma independiente al modelo arquitectónico orientado a objetos, es cierto que ha encontrado en este último una arquitectura base para estructurar los diferentes componentes que lo integran y las relaciones entre los mismos.

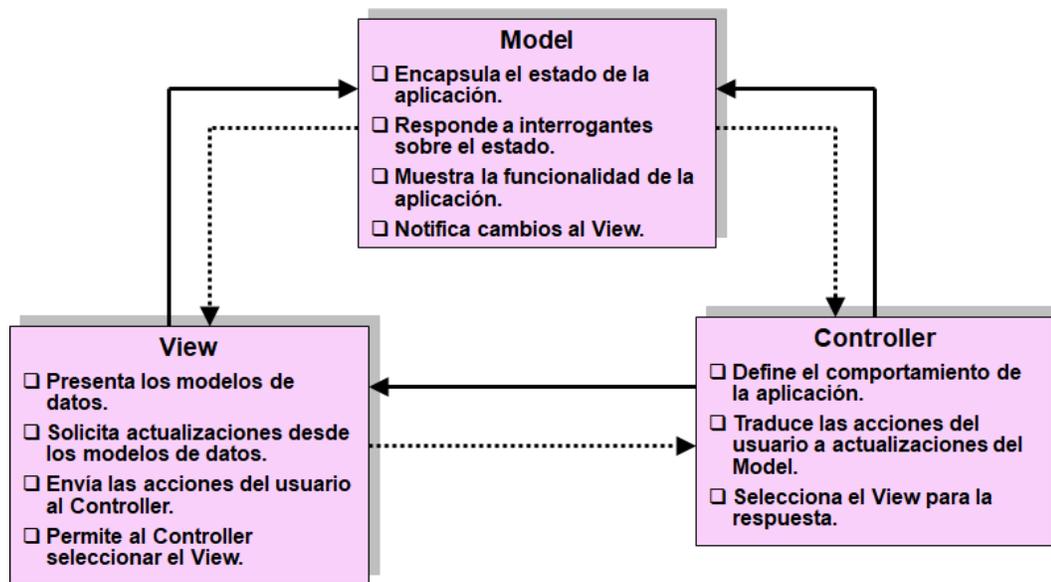
## V.2 Modelo arquitectónico de sistemas interactivos

Las complejas características del software a gran escala –incluyendo también la complejidad intrínseca de los requerimientos funcionales y no funcionales– exigen que el sistema de software sea diseñado y estructurado sobre una arquitectura que permita una verdadera separación e independencia entre aspectos de la vista, presentación e interacción, el modelo de datos, su acceso y almacenamiento, y el comportamiento, funcionalidades y procesamiento, exhibido por el sistema de software. La separación e independencia adecuadas entre estos tres aspectos clave de un sistema de software impactarán y repercutirán de forma positiva en todas las fases del desarrollo e ulteriores al desarrollo del mismo, desde la fase de análisis de requerimientos hasta la fase de verificación y pruebas y la ulterior etapa de mantenimiento.

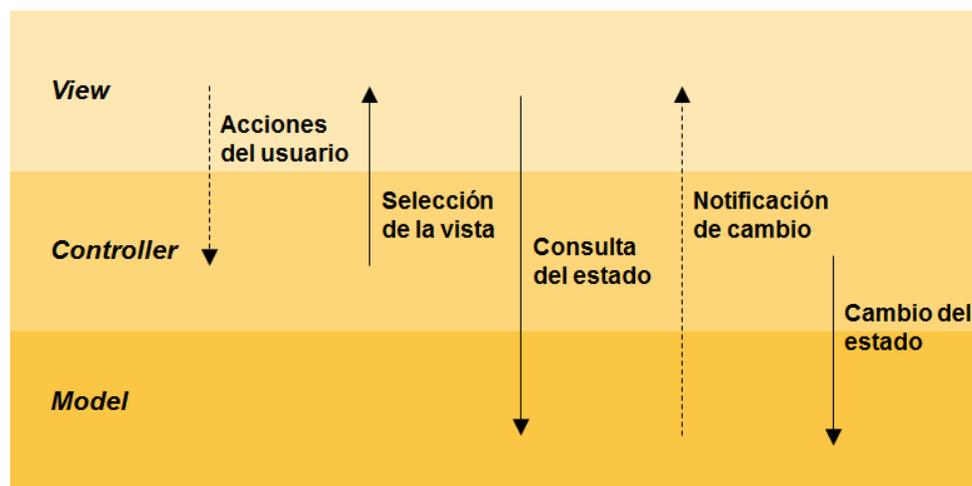
Atendiendo lo antes expuesto, en este epígrafe abordaremos de forma detallada la arquitectura que por excelencia proporciona la solución adecuada a las exigencias impuestas por el desarrollo y construcción de software a gran escala. Nos referimos a la arquitectura basada en el patrón arquitectónico Modelo-Vista-Controlador –del inglés, *Model-View-Controller* (MVC) [1-4].

El modelo de organización del software interactivo –comúnmente referido como arquitectura o patrón arquitectónico Modelo-Vista-Controlador (MVC), nace en el contexto del lenguaje orientado a objetos *Smalltalk* [5], con el objetivo de proporcionar una real separación entre aspectos de presentación y visualización de aspectos ligados a la lógica de la aplicación.

Según el modelo arquitectónico de sistemas interactivos, una aplicación interactiva se estructura separando la representación de los datos (modelo), la presentación de los datos (vista) y el comportamiento/lógica de la aplicación (controlador); tal como se ilustra en las Figuras 5.1 y 5.2.



**Figura 5.1** Componentes e Interacción en la Arquitectura Modelo-Vista-Controlador.



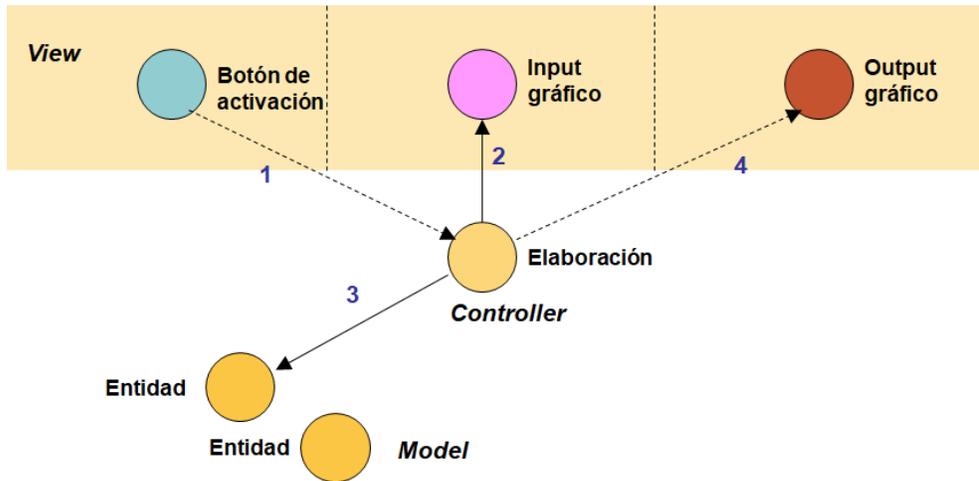
**Figura 5.2** Vista de Capas de la Arquitectura Modelo-Vista-Controlador.

**Modelo (Model).** Representa la estructura de los datos en la aplicación y las relativas operaciones (dependientes de la aplicación).

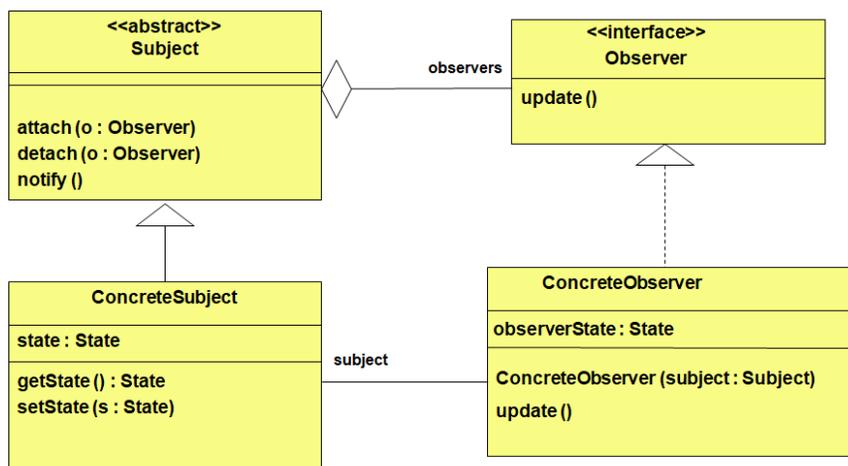
**Vista (View).** Es responsable de la interacción con el usuario. Presenta los datos en diferentes formas: puede haber más vistas en la medida en que sea necesario presentar los datos en formas diferentes. Recibe las entradas del usuario.

**Controlador (Controller).** Es sensible a las acciones del usuario, pudiendo recuperar los datos proporcionados por éste, trasladando los mismos en invocaciones de los métodos adecuados del modelo y seleccionando la vista apropiada (en base al estado y a la preferencia del usuario). En general, el controlador escucha los eventos de entrada que le llegan de la vista, reaccionando como consecuencia sobre el modelo. Funciona como coordinador entre la vista y el modelo.

La Figura 5.3 ilustra la dinámica del patrón arquitectónico MVC, mientras que la Figura 5.4 muestra el patrón *Observer*, el cual es comúnmente utilizado para el diseño e implementación de las fuentes generadoras de eventos y los oyentes de dichos eventos, en el MVC.



**Figura 5.3** Dinámica del patrón arquitectónico MVC.



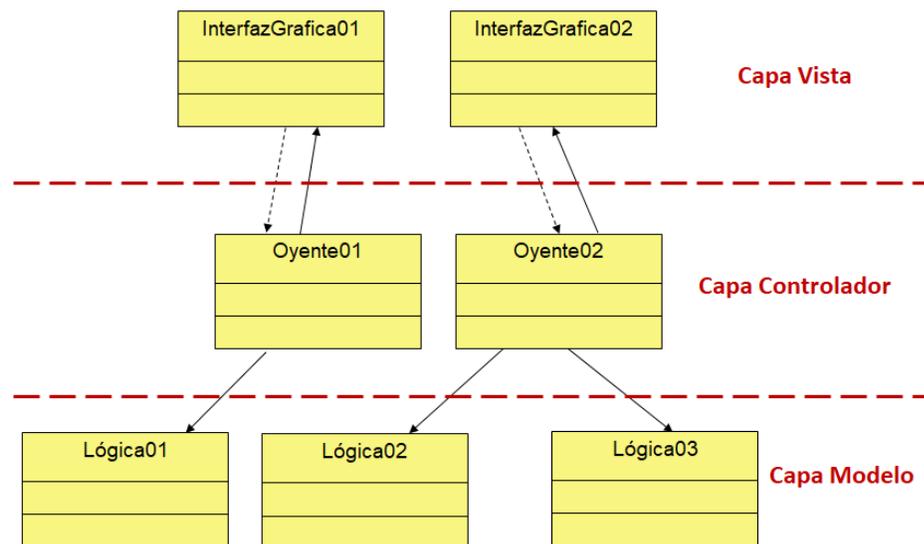
**Figura 5.4** Patrón *Observer*. Patrón comúnmente utilizado para el diseño e implementación de fuentes generadoras de eventos y oyentes de eventos en el MVC.

La interacción entre las fuentes generadoras de eventos en la capa Vista (*View*), los oyentes de dichos eventos en la capa Control (*Controller*), y los componentes de la lógica en la capa Modelo (*Model*), puede ser diseñada siguiendo varios enfoques, algunos de los cuales se ilustran en las Figuras 5.5 a la 5.7. Aquí cabe la pena hacer notar que estos enfoques son tanto aplicables a arquitecturas monolíticas como a arquitecturas cliente-servidor Web. En el caso de una arquitectura MVC Web desarrollada con Java, los elementos de la capa Control corresponden a los *Servlets*.

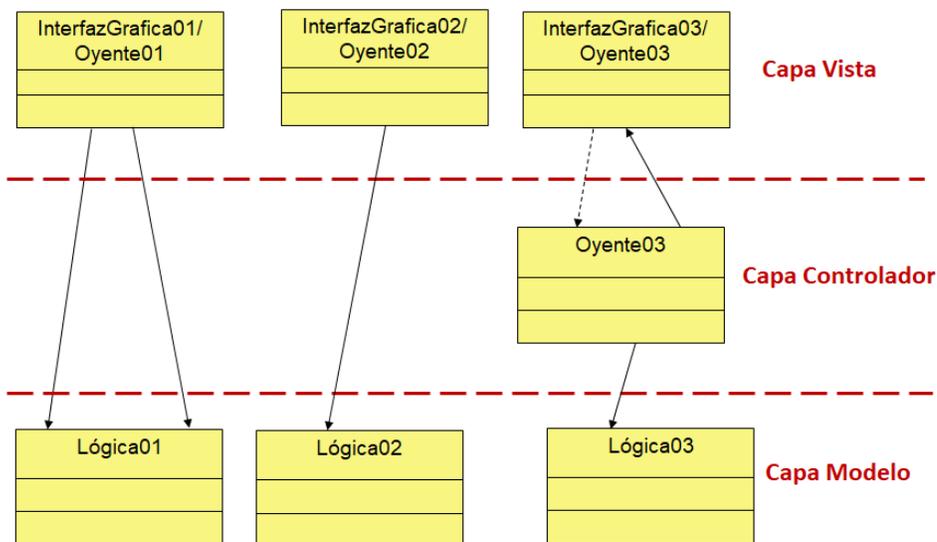
En la variante 1 (Figura 5.5), los oyentes son componentes que pertenecen a la capa Controlador. Comúnmente éstos no ejecutan ninguna función de la lógica de la aplicación, sólo reciben la notificación del evento ocurrido y en dependencia de la naturaleza del mismo, invocan los comportamientos necesarios de otros objetos de la capa Modelo, que representan parte de la lógica de la aplicación. Con este patrón se logra alcanzar un máximo desacoplamiento entre fuente generadora de eventos, oyentes y objetos de la lógica de la aplicación.

Por otra parte, en la variante 2 (Figura 5.6), el oyente es la propia interfaz gráfica en la capa Vista. Ésta se encarga de invocar los comportamientos necesarios de los componentes de la lógica (capa Modelo), en dependencia de la naturaleza del evento ocurrido. Con este patrón no se logra alcanzar un máximo desacoplamiento entre fuente generadora de eventos, oyentes y componentes de la lógica de la aplicación.

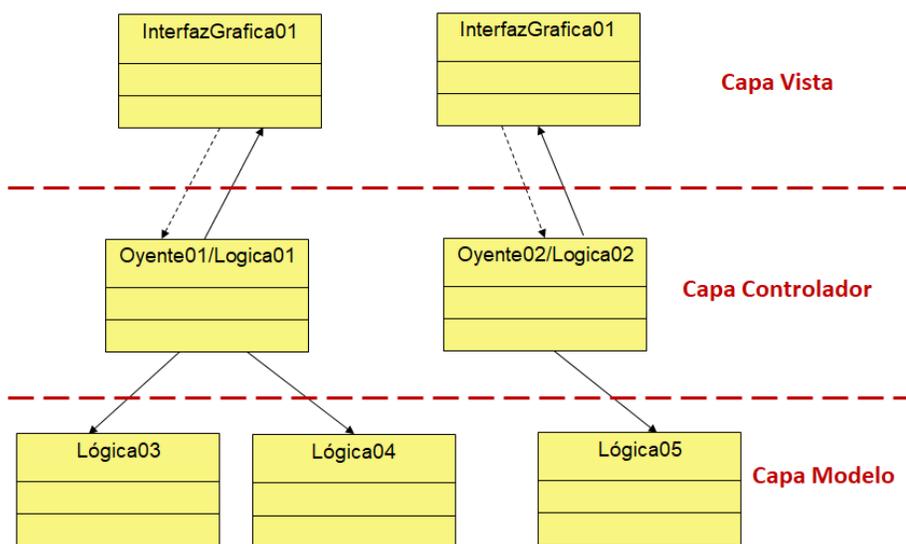
Finalmente, en la variante 3 (Figura 5.7), el oyente, además de recibir la notificación de los eventos ocurridos que interesan a él, puede también ejecutar determinados comportamientos de la lógica e interactuar con los componentes de la capa Modelo.



**Figura 5.5** Interacción entre los componentes de las capas Vista, Controlador y Modelo. Variante 1: los oyentes son objetos independientes que pertenecen a la capa Controlador, y actúan como intermediarios entre los objetos de la capa de la Vista y los objetos de la capa Modelo.



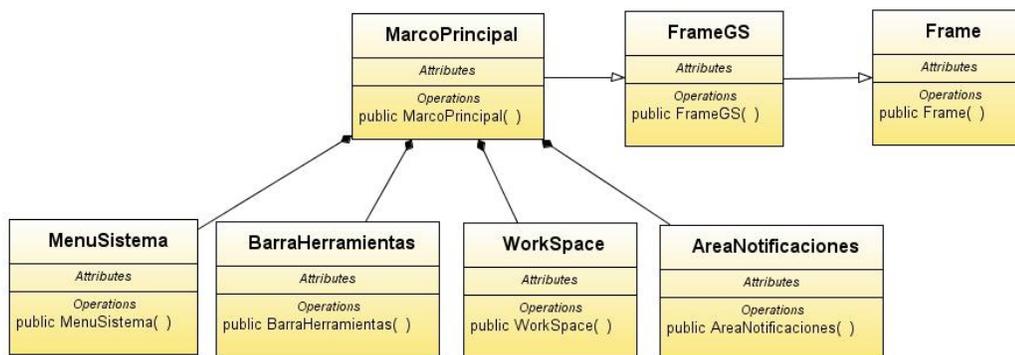
**Figura 5.6** Interacción entre los componentes de las capas Vista, Controlador y Modelo. Variante 2: el oyente es la propia interfaz gráfica en la capa Vista, tal como ocurre con los componentes InterfazGrafica01/Oyente01 e InterfazGrafica02/Oyente02.



**Figura 5.7** Interacción entre los componentes de las capas Vista, Controlador y Modelo. Variante 3: los componentes oyentes, además de escuchar los eventos que ocurren en las fuentes de la capa Vista, ejecutan algunas tareas de la lógica de la aplicación.

Las Figuras 5.8 a la 5.16 ilustran el uso de la arquitectura de sistemas interactivos MVC en diferentes aplicaciones desarrolladas por los autores.

En las Figuras 5.8, 5.9 y 5.10 se ilustran detalles de diseño arquitectónico del sistema RPS-GS [9, 10], una herramienta computacional fuertemente visual e interactiva, con las características propias de un ambiente de trabajo integrado (IDE, de la sigla en inglés), que proporciona al ingeniero de software un valioso y robusto soporte metodológico para la recuperación de proyectos de software abandonados o en peligro de abandono. La Figura 5.8 muestra un detalle del diseño de la capa Vista, con el contenedor *MarcoPrincipal* y sus componentes generadores de eventos *MenuSistema*, *BarraHerramientas*, *WorkSpace* y *AreaNotificaciones*. En la figura 5.9 se ilustra el diseño arquitectónico correspondiente a las capas Controlador y Modelo. Nótese que en esta figura todos los componentes de la capa Controlador son oyentes de los eventos que se generan en la capa Vista mostrada en la Figura 5.8. Finalmente, la Figura 5.10 exhibe un detalle de la arquitectura MVC del sistema RPS-GS, el módulo de gestión de documentos de las fuentes de información.



**Figura 5.8** Arquitectura MVC del Sistema de Recuperación de Proyectos de Software RPS-GS. Diseño de la capa Vista.

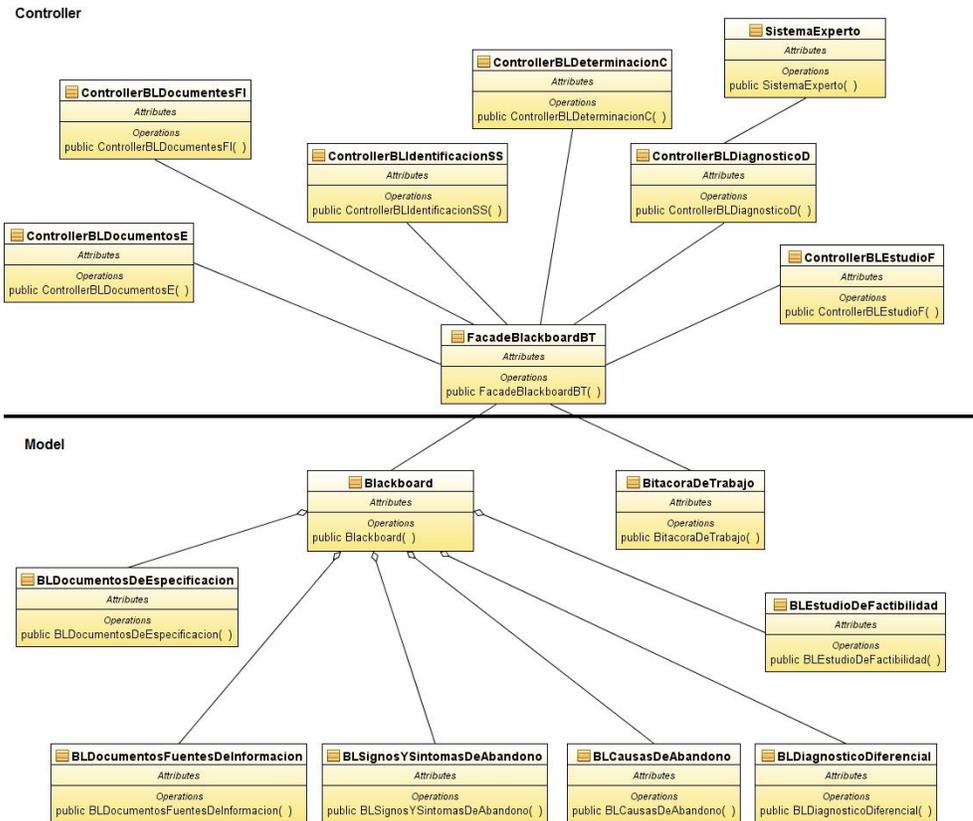
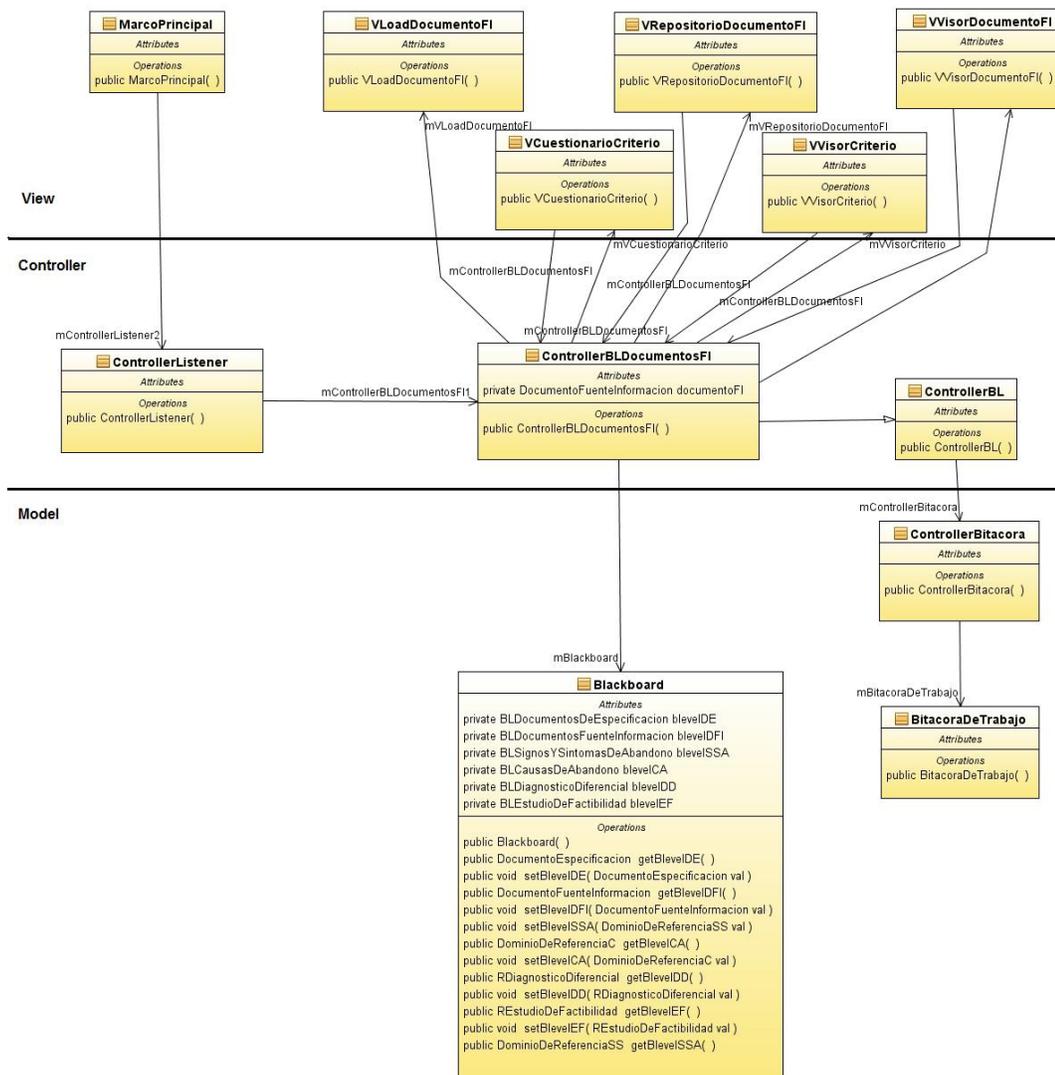
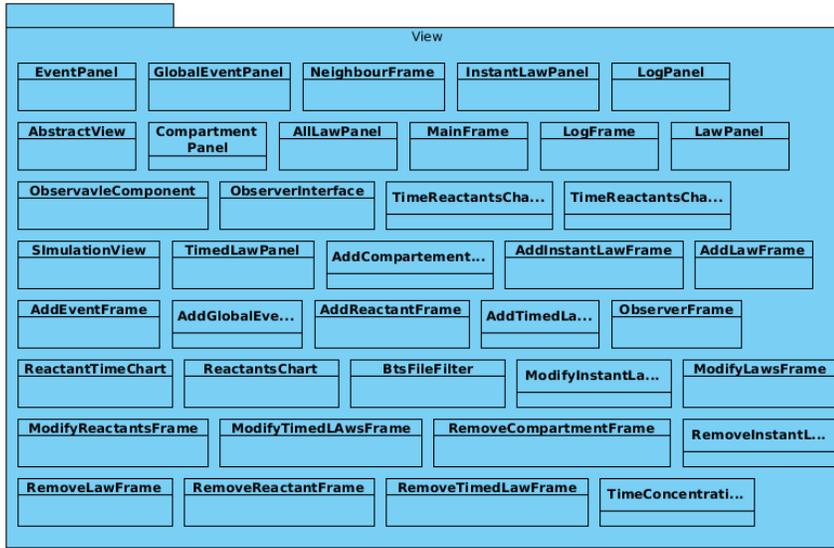


Figura 5.9 Arquitectura MVC del Sistema de Recuperación de Proyectos de Software RPS-GS. Diseño de las capas Controlador y Modelo.

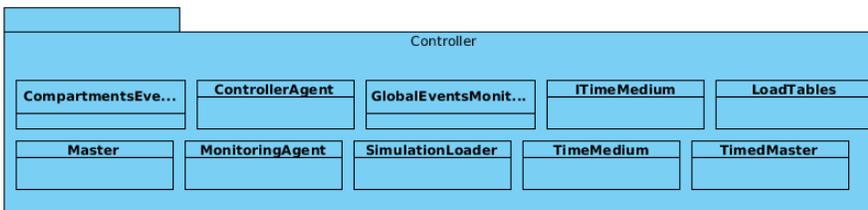


**Figura 5.10** Arquitectura MVC del Sistema de Recuperación de Proyectos de Software RPS-GS. Módulo: Gestión de Documentos Fuentes de Información.

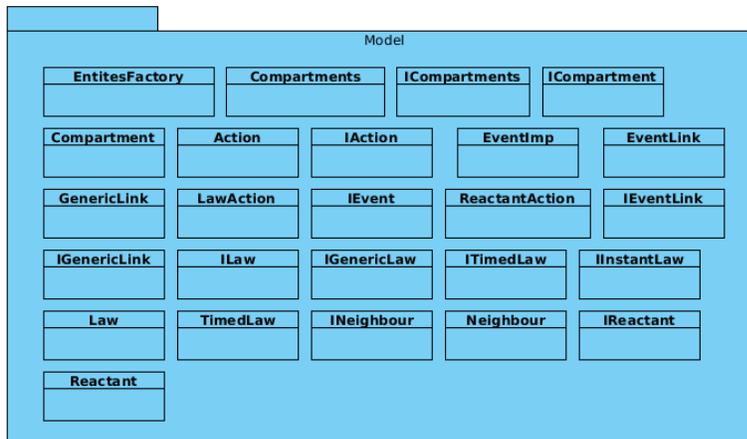
Las Figuras 5.11, 5.12 y 5.13 muestran la arquitectura MVC de la plataforma bioinformática Cellulat [11], una herramienta computacional altamente visual e interactiva para la simulación y experimentación *in silico* de redes de señalización celular. A diferencia de las Figuras. 5.8, 5.9 y 5.10, donde es posible apreciar las relaciones intracapa e intercapas, las Figuras 5.11, 5.12 y 5.13 ilustran la arquitectura MVC de Cellulat en términos de módulos o componentes.



**Figura 5.11** Arquitectura MVC de la Plataforma Bioinformática Cellulat. La capa Vista [11].

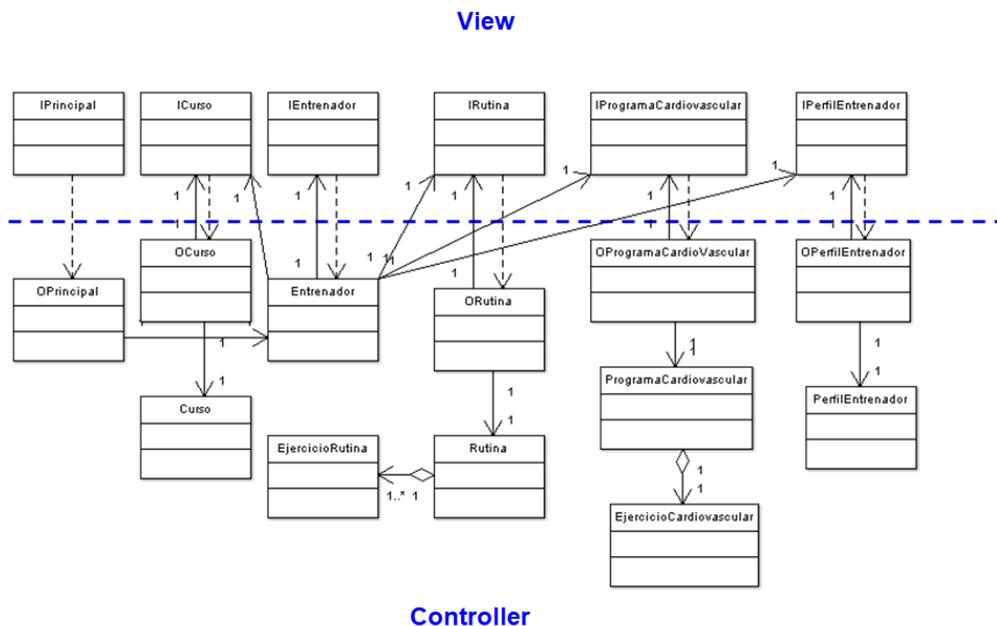


**Figura 5.12** Arquitectura MVC de la Plataforma Bioinformática Cellulat. La capa Controlador [11].

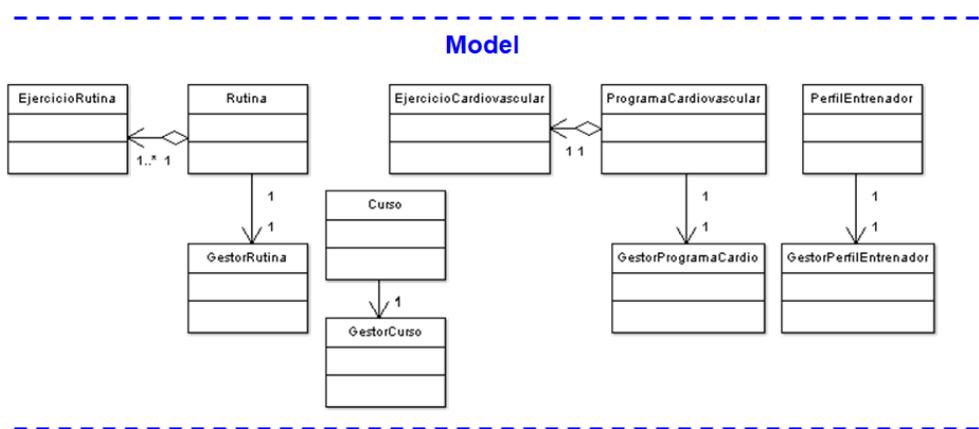


**Figura 5.13** Arquitectura MVC de la Plataforma Bioinformática Cellulat. La capa Modelo [11].

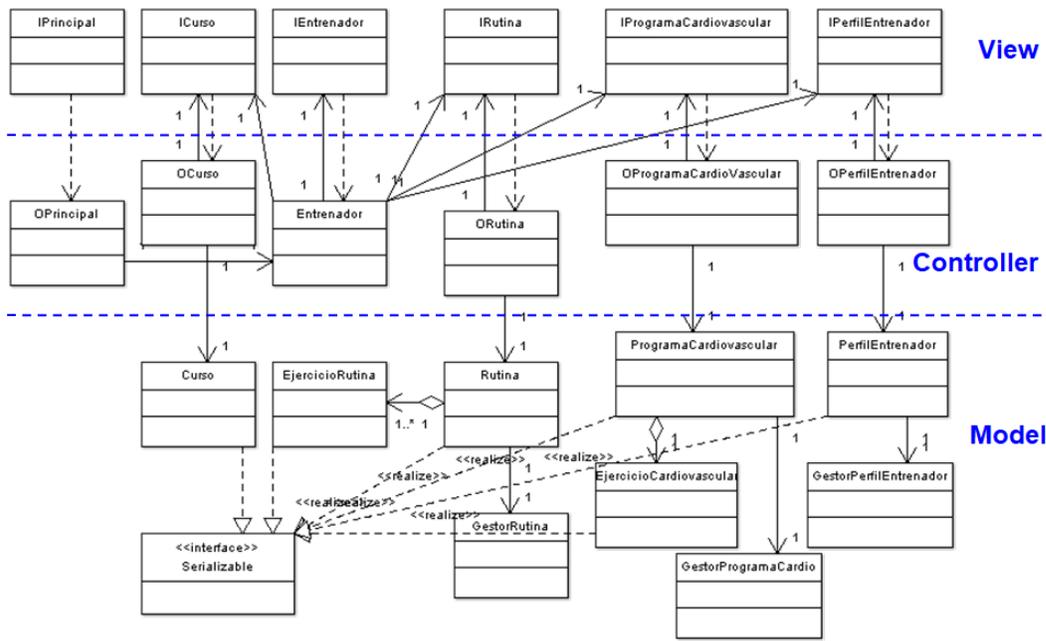
Por otra parte, las Figuras 5.14 a la 5.16 muestran aspectos del diseño arquitectónico del sistema Web Club Deportivo Virtual *Sport-Planet* [12]. La Figura 5.14 muestra la arquitectura del sistema *Sport Planet*, en términos de las capas Vista y Controlador y de las interacciones que toman lugar entre las mismas. La Figura 5.15 ilustra la capa Modelo, como parte del diseño arquitectónico de dicho sistema. Finalmente, la Figura 5.16 exhibe la arquitectura global del sistema interactivo *Sport Planet*, enfatizando las relaciones intracapa e intercapa que toman lugar.



**Figura 5.14** Arquitectura MVC del sistema Web Club Deportivo Virtual Sport Planet. Las capas Vista y Controlador.



**Figura 5.15** Arquitectura MVC del sistema Web Club Deportivo Virtual Sport Planet. La capa Modelo.



**Figura 5.16** Arquitectura MVC del sistema Web Club Deportivo Virtual Sport Planet. Relaciones entre las tres capas.

### V.3 Modelo Arquitectónico de Pizarra (*Blackboard Architecture*)

Los sistemas pizarra (blackboard systems) [6-8] fueron el primer intento de integrar módulos de software “cooperativos”. La meta era alcanzar el flexible estilo *brainstorming* de la solución de problemas exhibido por un grupo de diferentes expertos humanos trabajando conjuntamente para resolver un problema que un único experto no podía resolver por sí solo. La arquitectura de pizarra trata la solución de problemas como un proceso oportunista e incremental de ensamblaje de una configuración satisfactoria de elementos solución.

En la arquitectura de pizarra, el espacio de solución (todas las posibles soluciones parciales y globales del problema) está organizado en una o más jerarquías (dependientes de la aplicación). La información en cada nivel, dentro de una jerarquía, representa soluciones parciales al problema, existiendo en cada nivel un único vocabulario que describe la información. El conocimiento del dominio se encuentra particionado en módulos de conocimiento independientes, los cuales transforman la información de un nivel de jerarquía en información en el mismo nivel o en otros niveles.

Los módulos de conocimiento ejecutan estas transformaciones usando procedimientos algorítmicos o reglas heurísticas (IF condición THEN acción). El razonamiento oportunista que caracteriza a la arquitectura de pizarra se manifiesta

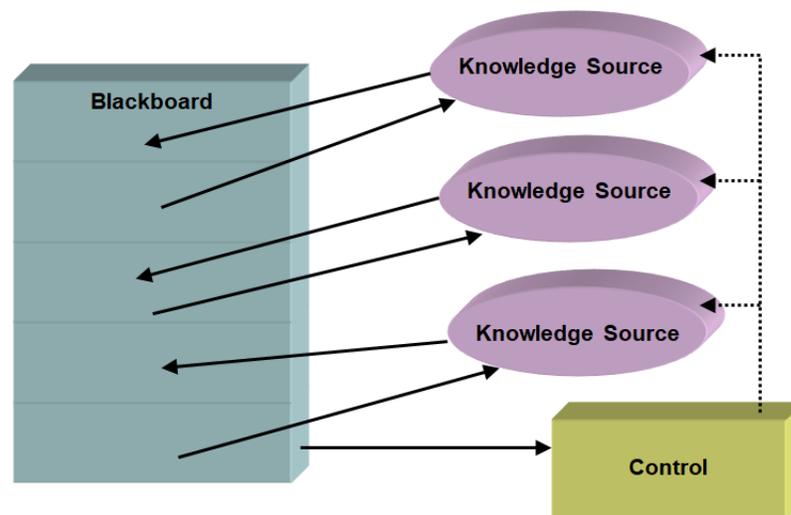
en los siguientes términos: en cada paso de la solución del problema, se determina dinámicamente qué módulo de conocimiento aplicar, resultando con ello una generación incremental de soluciones parciales.

La selección de un módulo de conocimiento específico se basa sobre el estado de la solución sobre la pizarra y sobre la existencia de módulos de conocimiento capaces de mejorar el estado actual de la solución. En cada paso de la solución del problema pueden ser aplicados métodos de razonamiento hacia adelante o hacia atrás existiendo, además, otros métodos de razonamiento tales como: direccionamiento por los eventos, direccionamiento por las metas, direccionamiento por los planes, direccionamiento por las expectativas, etc.

### V.3.1 Componentes de la arquitectura de pizarra

Como se puede apreciar en la Figura 5.17, la arquitectura de pizarra se estructura en base de tres componentes básicos [8]:

- Un conjunto de módulos independientes, llamados fuentes de conocimiento (*knowledge sources*) es decir, agentes, componentes, objetos, etc., que contienen conocimientos y habilidades específicos acerca del dominio del problema a resolver.
- Una pizarra (*blackboard*), repositorio o estructura de datos compartida, a través de la cual las fuentes de conocimiento se comunican entre sí.
- Un sistema de control (*control*), que determina el orden en que las fuentes de conocimiento operarán sobre los elementos solución creados sobre la pizarra.



**Figura 5.17** Componentes de la arquitectura de pizarra.

## Las fuentes de conocimiento

Las fuentes de conocimiento son conjuntos de conocimientos y habilidades representados como procedimientos, conjuntos de reglas o aserciones lógicas. Las fuentes de conocimiento poseen un formato condición-acción. La condición describe situaciones en las cuales la fuente de conocimiento puede contribuir a los procesos de solución del problema. Para que la condición de una fuente de conocimiento sea satisfecha, se requiere de una configuración particular de elementos solución sobre la pizarra. La acción se vincula a la creación o modificación de elementos solución sobre la pizarra.

De forma general, la actividad de las fuentes de conocimiento es dirigida por los eventos. Cada cambio sobre la pizarra constituye un evento que, en presencia de otra información específica disponible en la misma, puede disparar (satisfacer la condición de) una o más fuentes de conocimiento.

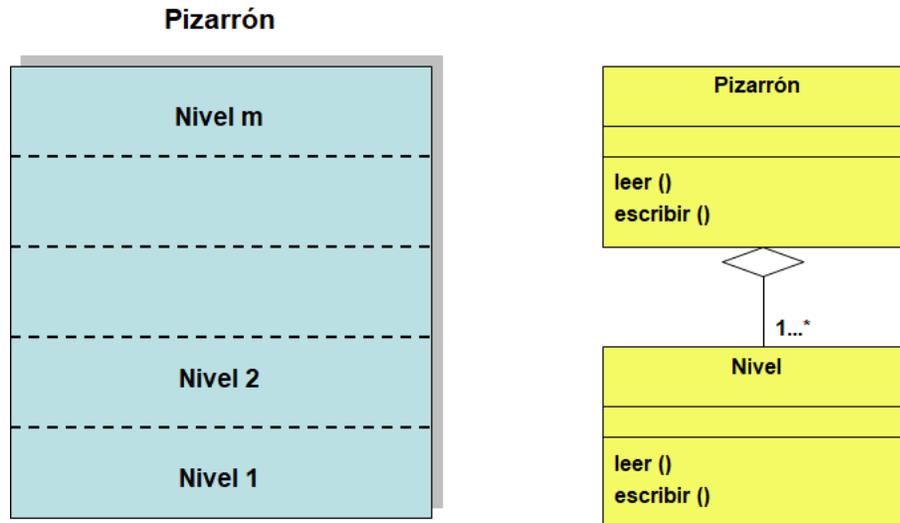
En términos de lenguajes de programación, una fuente de conocimiento puede ser implementada como una regla, en lenguajes de programación declarativos, tales como Prolog; como una función o un procedimiento, en lenguajes de programación estructurados, tales como C; o como un objeto, caracterizado por atributos y métodos, en lenguajes de programación orientados a objetos.

La arquitectura de pizarra garantiza independencia y cooperación simultánea entre las fuentes de conocimiento. Sin embargo, éstas son independientes en el sentido en que ellas mismas no se invocan unas a otras, y comúnmente no tienen conocimiento de la experticia, conocimiento o existencia de las otras fuentes de conocimiento. Son cooperativas en la medida que contribuyen elementos solución al problema compartido.

## La pizarra o repositorio

Los datos correspondientes al estado de la solución del problema son colocados en un repositorio global, denominado pizarra. Las fuentes de conocimiento producen cambios sobre la pizarra, los cuales conducen incrementalmente a la formación de una solución o conjunto aceptable de soluciones para el problema que se intenta resolver. La interacción entre las fuentes de conocimiento es indirecta y ésta sólo toma lugar a través de los cambios producidos sobre la pizarra.

La pizarra consiste de elementos solución pertenecientes al espacio solución del problema. Estos elementos solución pueden ser datos de entrada, soluciones parciales, soluciones alternativas o soluciones finales. Como se muestra en la Figura 5.18, los elementos solución se encuentran organizados sobre la pizarra en varios niveles de análisis o de abstracción. Los diferentes niveles de abstracción representan la solución del problema en diferente cantidad de detalle o de contexto.



**Figura 5.18** La organización de la pizarra en niveles. En la parte derecha se ilustra parte del patrón arquitectónico.

### El mecanismo de control

Debido a que las fuentes de conocimiento responden de forma oportuna a cambios sobre la pizarra, es necesario un mecanismo de control que monitoree estos cambios y decida, en cada momento, cuáles acciones se deben tomar. El mecanismo de control utiliza varios tipos de información de gran utilidad, que se puede encontrar sobre la pizarra o fuera de ésta, para así determinar el llamado “foco de atención”.

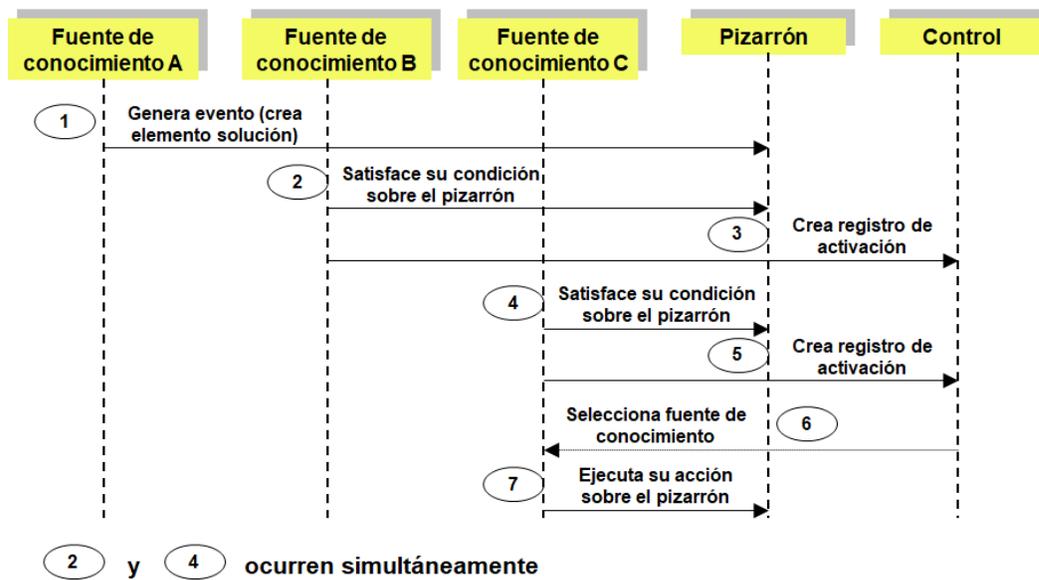
El foco de atención indica que será lo próximo a ser procesado. Éste puede contener información referente a una fuente de conocimiento (por ejemplo, la próxima fuente de conocimiento a activar); a un elemento solución sobre la pizarra (por ejemplo, cuál elemento solución separar para dedicarse a éste próximamente) o a una combinación de ambos (cuáles fuentes de conocimiento aplicar a cuáles elementos solución). La gran mayoría de los sistemas utilizan sólo una de estas tres posibilidades.

En el enfoque tradicional de la arquitectura de pizarra, durante el proceso de solución del problema, las fuentes de conocimiento y el mecanismo de control ejecutan sus actividades en la siguiente secuencia iterativa:

1. Ocurrencia de un evento sobre la pizarra: una fuente de conocimiento ha generado o modificado un elemento solución sobre la pizarra. Esta acción también ha producido un registro en una estructura de datos con información de control.
2. Cada una de las fuentes de conocimiento informa la contribución que puede hacer al nuevo estado solución.

- Teniendo en cuenta la información contenida en 1 y 2, el mecanismo de control selecciona un foco de atención.

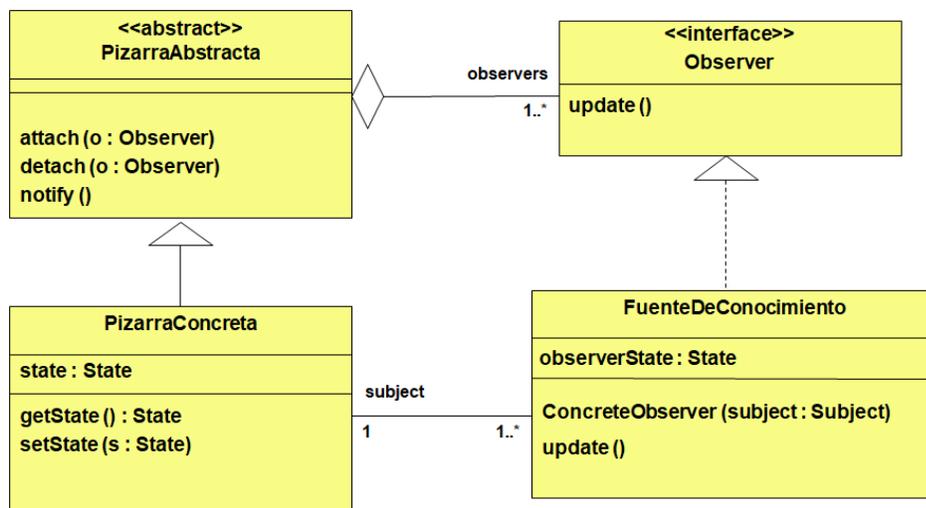
La Figura 5.19 ilustra la anterior secuencia iterativa, a través de un diagrama de secuencia.



**Figura 5.19** Ejecución de actividades por parte de las fuentes de conocimiento y el mecanismo de control.

Sin embargo, cuando la arquitectura de pizarra es concebida, diseñada e implementada siguiendo el enfoque de sistemas interactivos, entonces el control podría reducirse a la dinámica del patrón Observer, ya introducido en la arquitectura de sistemas interactivos. Es decir, con este enfoque las fuentes de conocimiento no deben iterar un ciclo de lectura sobre la pizarra en espera de la ocurrencia de un evento de interés; sino que éstas son notificadas por la pizarra (la cual funge ahora como fuente generadora de eventos) en el momento en que un evento de interés (un evento para el cual una fuente de conocimiento se registra como oyente del mismo) haya ocurrido.

La Figura 5.20 ilustra el uso del patrón Observer, en el diseño de la arquitectura de pizarra como una arquitectura de sistema interactivo.



**Figura 5.20** El patrón Observer utilizado como elemento de control en la arquitectura de pizarra.

Las Figuras 5.21 y 5.22 ilustran un escenario en el cual, la naturaleza oportunista e incremental de la solución del problema particular, encontró un adecuado enfoque en la arquitectura de pizarra. Nos referimos al sistema de recuperación de proyectos RPS-GS [9, 10], al cual ya nos hemos referido en capítulos previos. La Figura 5.21 muestra el modelo de pizarra con niveles de abstracción y fuentes de conocimientos requeridos para el problema de recuperación de proyectos de software. Por otra parte, la Figura 5.22 muestra la arquitectura MVC, solo a nivel de las capas Controlador y Modelo, del sistema RPS-GS. Nótese en esta figura como la arquitectura lógica del sistema RPS-GS combina aspectos de la arquitectura de sistemas interactivos con aspectos de la arquitectura de pizarra en un mismo modelo arquitectónico. Como se puede observar en esta figura, la capa Modelo se define en términos de la arquitectura de pizarra. Finalmente, la Figura 5.23 muestra la arquitectura MVC del sistema RPS-GS, incluyendo las tres capas Vista, Controlador y Modelo. Por simplicidad, en esta figura no han sido incluidas todas las clases que conforman cada una de las capas de la arquitectura MVC.

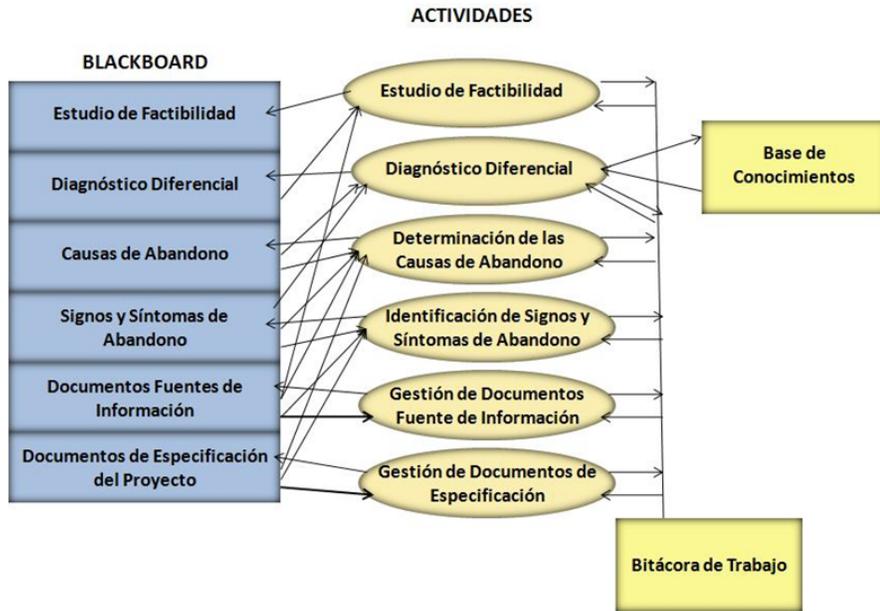


Figura 5.21 Modelo de pizarra del sistema de recuperación de proyectos de software RPS-GS.

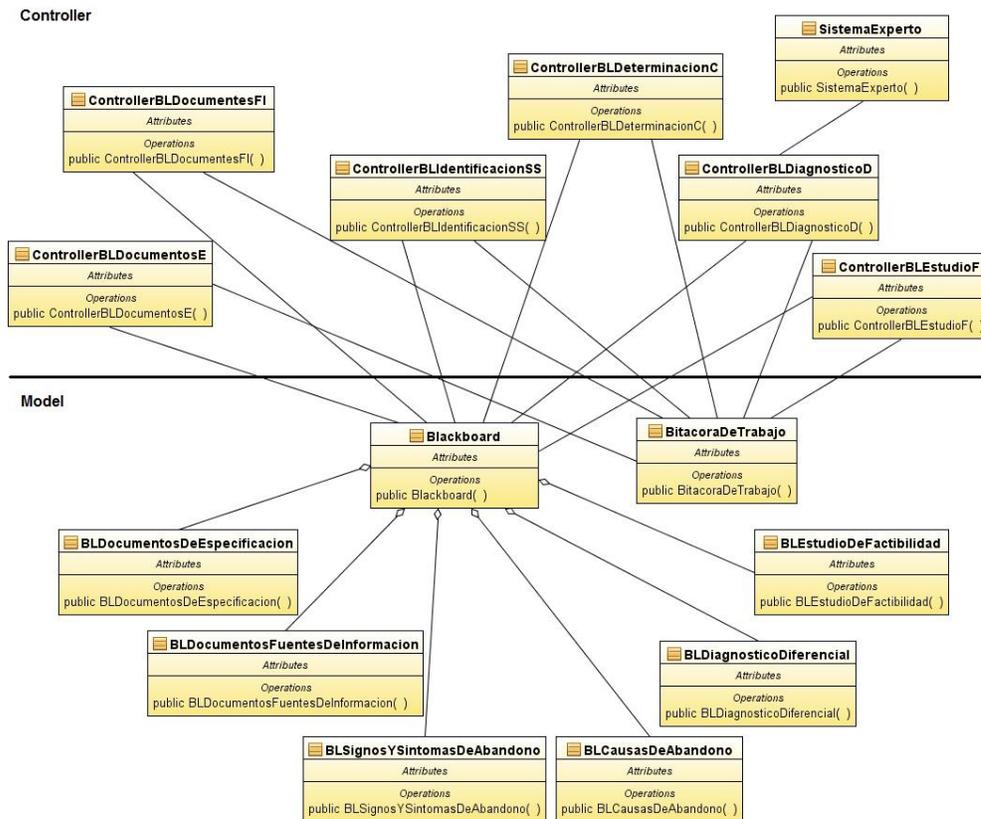
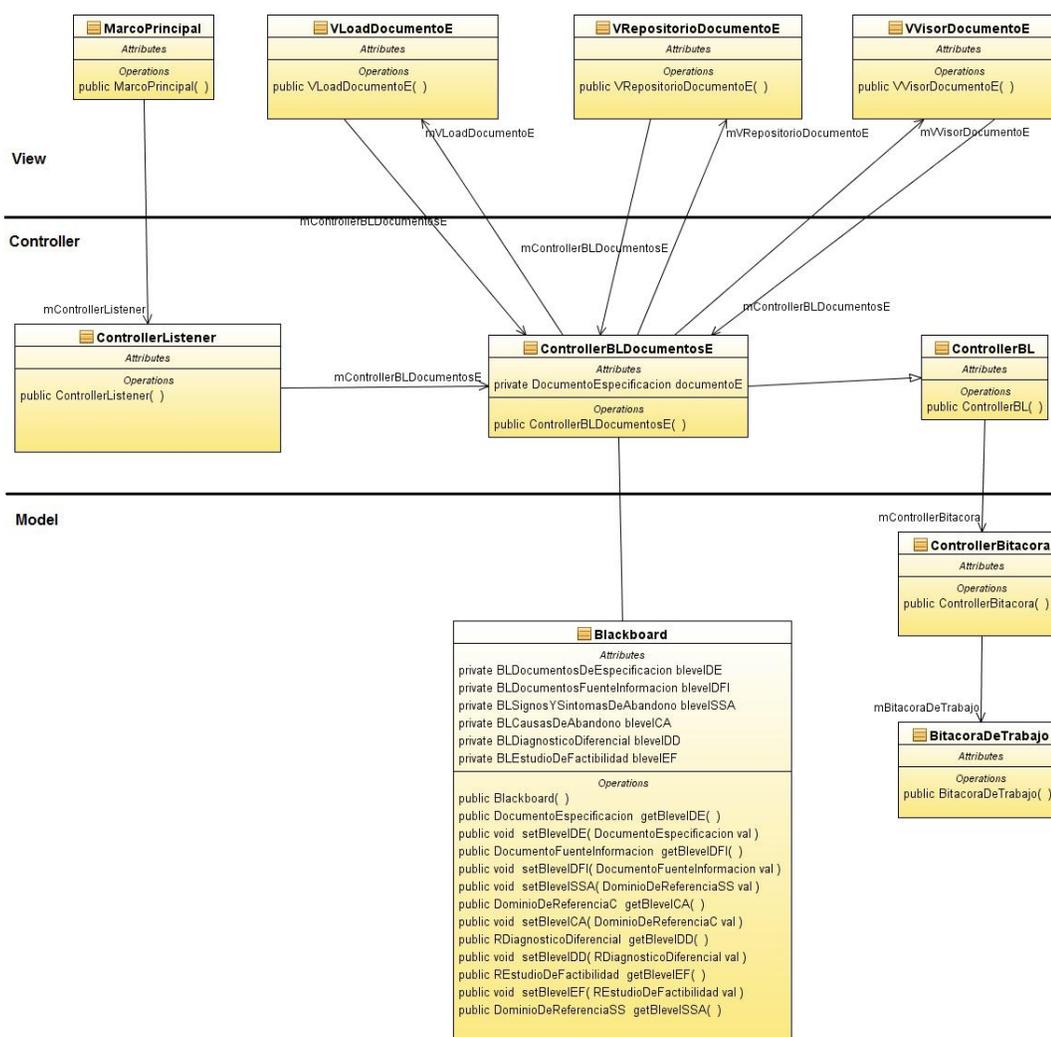


Figura 5.22 Arquitectura MVC sistema de recuperación de proyectos de software RPS-GS, mostrando solo los niveles Controlador y Modelo.



**Figura 5.23** Arquitectura MVC sistema de recuperación de proyectos de software RPS-GS, mostrando los tres niveles Vista Controlador y Modelo. Nótese que por simplicidad no se han incluido todas las clases de las capas Controlador y Modelo exhibidas en la Figura 5.22.

### V.4 La Arquitectura Física del Software

La arquitectura física del software se refiere a la implementación de la arquitectura lógica en los componentes de hardware requeridos, es decir procesadores, nodos, computadoras, o cualquier otro tipo de hardware al que sea asignada alguna de las tareas o funcionalidades que deberá llevar a cabo el sistema software.

Entre las principales arquitecturas físicas del software se encuentran la siguientes:

Diseño Arquitectónico y Diseño de la Interacción entre Interfaces Gráficas de Usuario  
— La Arquitectura Física del Software

- Sistemas monolíticos (del inglés, *standalone*).
- Sistemas Distribuidos.
- Sistemas Cliente-Servidor Web.

Como se puede apreciar en la Figura 5.24, toda arquitectura lógica requiere ser asignada a una arquitectura física, de forma tal que el software que expresa la arquitectura lógica pueda ser ejecutado sobre los componentes físicos que definen la arquitectura física. En otras palabras, los subsistemas, módulos, componentes, etc., que integran la solución expresada a través de la arquitectura lógica, deben ser asignados a los componentes físicos, tales como, un único procesador, un cliente y un servidor, un servidor web, un cluster de nodos, etc., requeridos en la solución del problema. La adecuada selección de la arquitectura física es un aspecto crucial en el desarrollo de software a gran escala. Comúnmente, el tipo de arquitectura lógica adoptada induce el tipo de arquitectura física a utilizar.

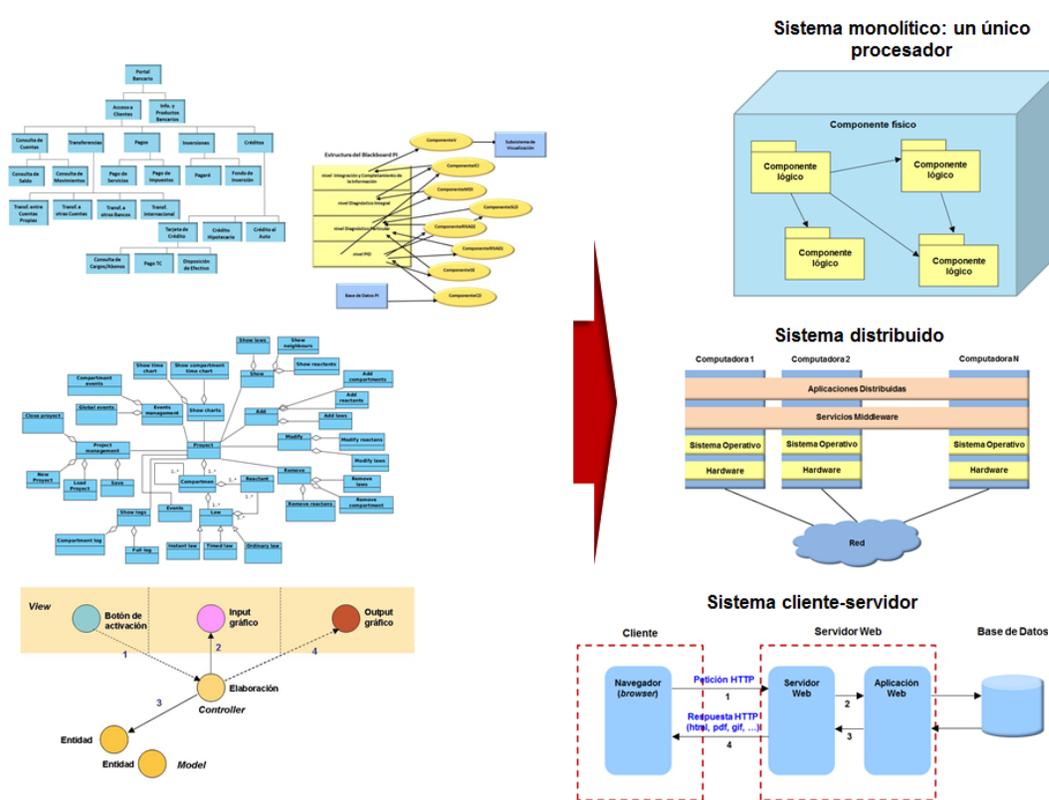
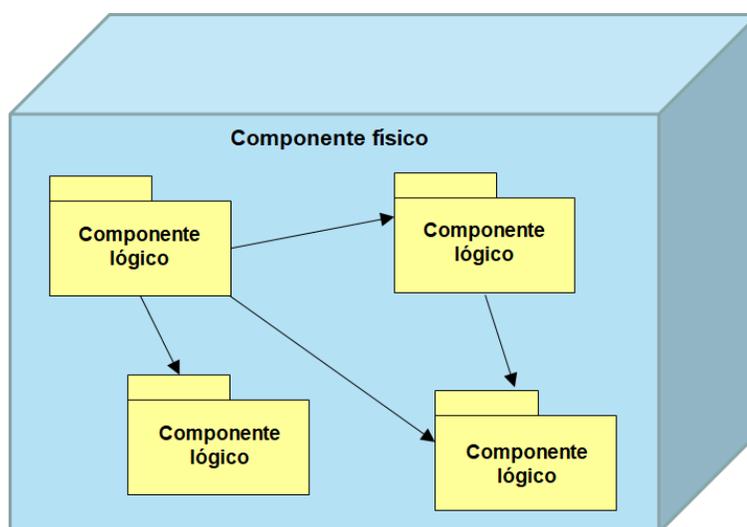


Figura 5.24 De la arquitectura lógica a la arquitectura física.

### V.4.1 Sistemas Monolíticos

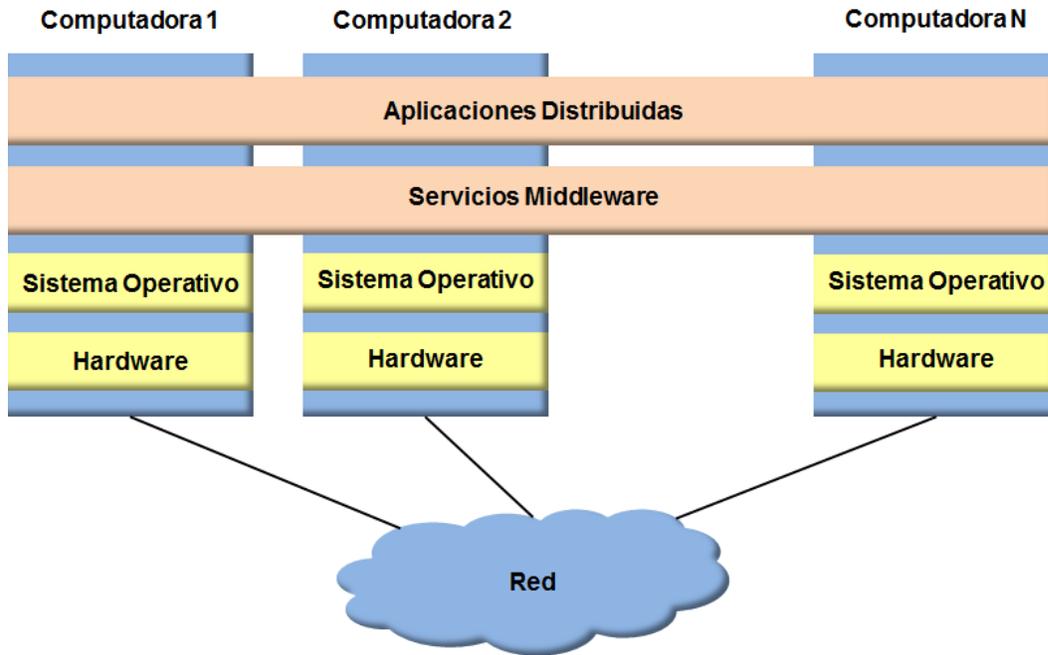
En una arquitectura monolítica, todos los componentes lógicos (funciones, objetos, componentes, módulos, subsistemas, etc.) y de bases de datos son asignados a un único procesador físico. Es decir, todos los componentes lógicos que participan en la solución del problema se asignan a un único componente de hardware. En este tipo de arquitectura no se requieren canales de comunicación físicos. De esta forma, cualquier sistema de software cuya arquitectura lógica (componentes e interacciones entre los componentes) sea completamente asignada a un único procesador físico, exhibirá una arquitectura física del tipo sistema monolítico. La Figura 5.25 ofrece una representación esquemática de la arquitectura monolítica.



**Figura 5.25** Asignación de componentes lógicos a un único componente físico en la arquitectura monolítica.

### V.4.2 Sistemas Distribuidos

Una arquitectura de sistemas distribuidos involucra varios nodos hardware (principalmente procesadores) y los componentes lógicos (funciones, clases, módulos, subsistemas, etc.) se encuentran asignados a dichos nodos. Los nodos y los componentes lógicos se comunican y coordinan sus actividades a través del paso de mensajes. Una característica principal de este tipo de sistemas es la concurrencia, mientras que el control y el procesamiento se encuentra distribuido entre los nodos. En la Figura 5.26 se ofrece una representación esquemática de la arquitectura de sistema distribuido.



**Figura 5.26** Representación esquemática de la arquitectura de sistemas distribuidos.

### V.4.3 Sistemas Cliente-Servidor Web

La arquitectura de un sistema cliente-servidor estructura una aplicación como uno o más servidores que proporcionan servicios y un conjunto de clientes que acceden y usan los servicios. Desde el punto de vista físico, los clientes pueden ser diferentes tipos de hardware, tales como computadoras de escritorios, laptops, dispositivos móviles, etc., mientras que los servidores son comúnmente potentes equipos de cómputo, caracterizados por grandes recursos de memoria y velocidad de procesamiento.

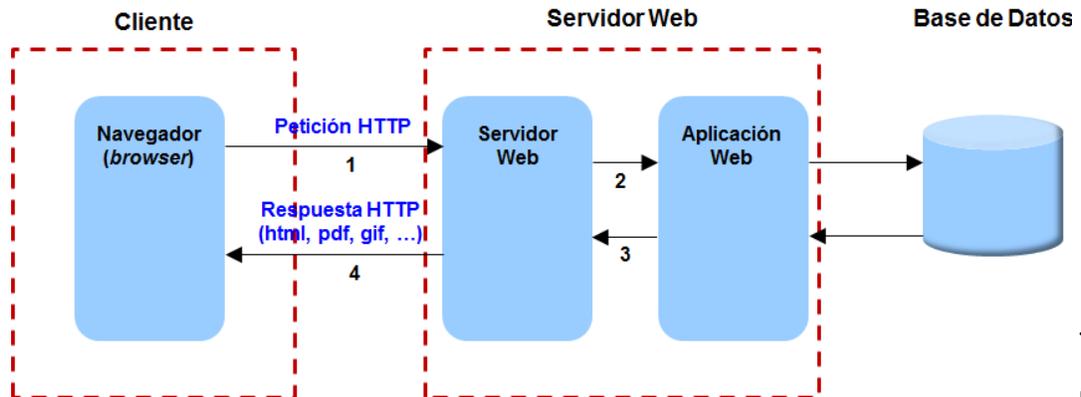
Desde el punto de vista lógico, los clientes son los navegadores (browsers) que se ejecutan en el dispositivo físico cliente, mientras que los servidores se refieren a todos los servicios proporcionados por servidores web, servidores de aplicaciones, servidores de bases de datos, entre otros. Es decir, el software que se ejecuta en el servidor físico.

La arquitectura cliente servidor puede ser vista como una arquitectura distribuida, que permite a los clientes acceder a los recursos que proporcionan los servidores. La interacción entre el cliente y el servidor se da a través de dos tipos principales de mensajes:

- Petición: mensaje enviado por el cliente al servidor, solicitando un determinado servicio.

- Respuesta: mensaje enviado por el servidor al cliente, proporcionando el servicio solicitado.

Entre clientes y servidores media una estructura de comunicación, conocida como red. En aplicaciones cliente-servidor web, dicha red es Internet. La Figura 5.27 ofrece una representación esquemática de la arquitectura de sistema cliente-servidor Web.



**Figura 5.27** Representación esquemática de la arquitectura de un sistema cliente-servidor web.

Desde el punto de vista lógico, el cliente puede ser visto como el software que permite al usuario formular las peticiones y enviarlas como mensajes al servidor. Esta parte lógica es comúnmente conocida como front-end. En una arquitectura cliente-servidor, el cliente comúnmente maneja toda la funcionalidad relacionada con la interacción y visualización. Entre las principales funcionalidades que lleva a cabo la parte lógica cliente (front-end) se encuentran:

- Establecimiento de todo tipo de interacción con el usuario.
- Proporcionar las interfaces gráficas (GUI), a través de las cuales el usuario efectúa sus peticiones al servidor, registra información, realiza selecciones, etc.
- Generar requerimientos a bases de datos.
- Formatear, presentar y visualizar las respuestas enviadas por el servidor.

Desde el punto de vista lógico, el servidor proporciona el software que se encarga de responder a las peticiones efectuadas por los clientes, ya sean relacionadas con la lógica de la aplicación o bases de datos. Es decir, desde el punto de vista lógico, es precisamente el servidor quien maneja todas las funcionalidades relacionadas con la lógica de la aplicación (reglas de negocio) y con las bases de datos. La parte lógica servidor es comúnmente conocida como

back-end. Entre las principales funcionalidades que lleva a cabo la parte lógica servidor (back-end) se encuentran:

- Procesar la lógica de la aplicación, para atender peticiones efectuadas por el cliente.
- Preparar la respuesta de requerimientos de la lógica de la aplicación y enviarla al cliente.
- Recibir y procesar los requerimientos de bases de datos que efectúa el cliente.
- Preparar la respuesta de requerimientos de bases de datos y enviarla al cliente.

## V.5 Vistas de la arquitectura del software

La arquitectura del software viene comúnmente representada desde varias perspectivas (lógica, de proceso, de despliegue, de datos, etc.) lo cual permite una mejor comprensión del diseño. Cada perspectiva constituye una vista diferente de la arquitectura del software, mientras que todas las vistas juntas describen la arquitectura. En el desarrollo de software a gran escala, resulta muy valioso comprender la arquitectura propuesta a través de sus aspectos estáticos, lógicos, de procesos y físicos, entre otros. En este sentido, el contar con un apropiado modelo de vistas de la arquitectura, proporciona escenarios adecuados para validar la arquitectura del software propuesta tomando en consideración los aspectos antes mencionados. Aquí haremos referencia a dos de los modelos de vistas de arquitectura comúnmente usados en el desarrollo de software orientado a objetos:

- El Modelo de Vistas 4+1 (The 4+1 View Model of Architecture) [13].
- El Modelo de Vistas del Proceso Unificado de Desarrollo de Software [14].

Ambos modelos de vistas de la arquitectura del software serán ilustrados en el epígrafe V.9, a través de los dos casos de estudio ya utilizados en los anteriores epígrafes, y con los cuales seguramente ya el lector se ha familiarizado, SPI y Evolution.

### V.5.1 El modelo de vistas 4+1

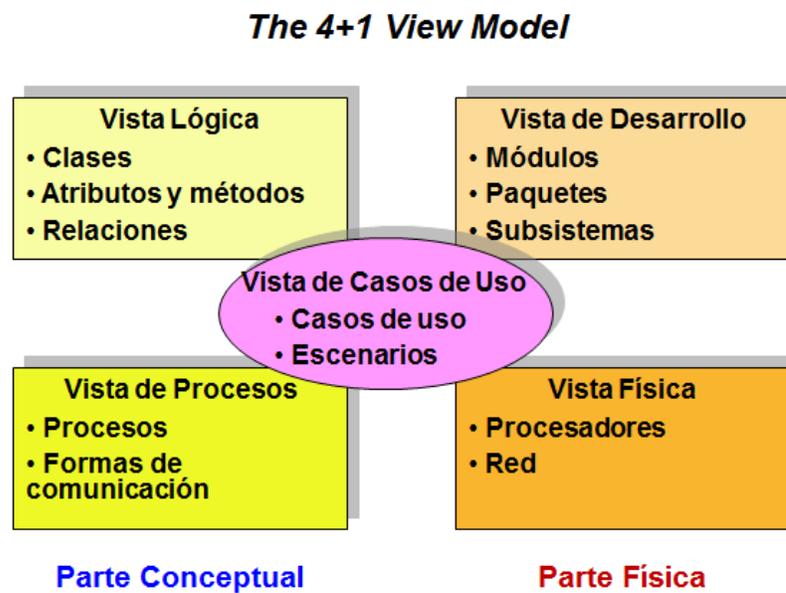
Como se ilustra en la Figura 5.28, el Modelo de Vistas 4+1 [13] permite describir la arquitectura de un sistema de software a partir de 4 vistas esenciales:

- Vista lógica.
- Vista de proceso.

- Vista de desarrollo.
- Vista física.

Una quinta vista es proporcionada por el modelo 4+1: la vista de casos de uso, la cual describe los aspectos funcionales del sistema como un todo. Mientras que la vista lógica y la vista de procesos describen la parte conceptual del sistema de software, la vista de desarrollo y la vista física describen la parte física del sistema. El Modelo de Vistas 4+1 puede ser incrementado de forma tal que incluya las vistas correspondientes a los requerimientos no funcionales del sistema, tales como: calidad, seguridad, portabilidad, etc.

El Modelo de Vistas 4+1 resulta muy adecuado para la descripción de la arquitectura de sistemas de software a gran escala, debido al gran nivel de detalle que proporciona sobre cada uno de estos importantes aspectos (vistas) de la arquitectura de un sistema de software. La Tabla 5.1 resume los aspectos, elementos y modelos que encierra cada una las vistas del Modelo de Vistas 4+1.



**Figura 5.28** El Modelo de Vistas 4+1.

**Tabla 5.1.** Aspectos, elementos y modelos que abarcan las vistas del Modelo de Vistas 4+1.

Vista	Aspectos que focaliza	Elementos que captura	Modelos/artefactos usados
<b>Vista Lógica</b>	Modelo del dominio Requerimientos de comportamiento Principales conceptos del dominio	Clases Atributos y comportamientos Interfaces Relaciones entre clases	Diagramas de clases
<b>Vista de Procesos</b>	Comunicación Concurrencia Sincronización	Objetos Procesos Hilos ( <i>Threads</i> )	Diagramas de clases Diagramas de interacción Diagramas de actividad Casos de uso
<b>Vista de Desarrollo</b>	Módulos del sistema	Subsistemas Paquetes Bibliotecas de clases Organización de archivos	Diagramas de paquetes Diagramas de clases
<b>Vista Física</b>	Distribución del software y del hardware Instalación y ejecución de la aplicación en un ambiente distribuido	Procesadores Nodos Redes	Diagramas de componentes Diagramas de despliegue ( <i>deployment</i> )
<b>Vista de Casos de Uso</b>	Funcionalidad del sistema Descripción y consolidación de las restantes vistas	Casos de uso Escenarios	Casos de uso Diagramas de secuencia

### V.5.2 El modelo de vistas del proceso unificado de desarrollo de software

El Modelo de Vistas del Proceso Unificado (UP) [14] comparte varios de los aspectos, elementos y modelos presentes en el Modelo de Vistas 4+1, proporcionando 6 vistas para describir y comprender la arquitectura de un sistema de software:

- Vista Lógica.
- Vista de Proceso.
- Vista de Despliegue.
- Vista de Datos.
- Vista de Casos de Uso.
- Vista de Implementación.

Nótese que la Vista de Datos es la vista adicional que proporciona este modelo de vistas en comparación con el Modelo de Vistas 4+1, ya que las vistas de Despliegue y de Implementación corresponden prácticamente a la vista Física del Modelo de Vistas 4+1.

Al igual que el Modelo de Vistas 4+1, el Modelo de Vistas del Proceso Unificado resulta de gran utilidad para la descripción de la arquitectura de sistemas de software a gran escala, debido al gran nivel de detalle que proporciona sobre cada uno de estos importantes aspectos (vistas) de la arquitectura de un sistema de software. La Tabla 5.2 resume los aspectos, elementos y modelos que encierra cada una las vistas del Modelo de Vistas del Proceso Unificado.

**Tabla 5.2.** Aspectos, elementos y modelos que abarcan las vistas del Modelo de Vistas del Proceso Unificado.

Vista	Aspectos que focaliza	Elementos que captura	Modelos/artefactos usados
<b>Vista Lógica</b>	Organización conceptual del software Funcionalidad de los elementos más importantes del software	Capas Subsistemas Paquetes Marcos de referencia ( <i>Frameworks</i> ) Clases Relaciones Interfaces	Diagramas de clases Diagramas de paquetes
<b>Vista de Proceso</b>	Comunicación Concurrencia Sincronización	Objetos Procesos Hilos ( <i>Threads</i> )	Diagramas de clases Diagramas de interacción Diagramas de actividades
<b>Vista de Despliegue</b>	Distribución del software y del hardware Despliegue físico de los procesos y componentes sobre los nodos y la configuración de la red física entre los nodos	Procesos Procesadores Nodos Redes	Diagrama de componentes Diagrama de despliegue ( <i>deployment</i> )
<b>Vista de Datos</b>	Vista global del modelo de datos persistente Correspondencia de objetos a datos persistentes	Objetos datos Objetos gestores de datos Base de datos	Tablas y relaciones Diagramas de clases Diagramas Entidad-Relación Diagramas del Modelo Relacional Diagramas de Bases de Datos Orientadas a Objetos
<b>Vista de casos de uso</b>	Casos de uso más significativos para la arquitectura y sus requerimientos no funcionales	Casos de uso Actores Dependencias entre casos de uso	Descripción textual de los casos de uso (scripts) Diagramas de casos de uso
<b>Vista de implementación</b>	Descripción resumida de la organización relevante de los entregables y de los elementos que crean los entregables	Código fuente Páginas Web DLLs (Dynamic Linking Library) Ejecutables Archivos JAR	Diagramas de paquetes y componentes

Diseño Arquitectónico y Diseño de la Interacción entre Interfaces Gráficas de Usuario  
— Vistas de la arquitectura del software

## V.6 Interacciones entre las interfaces de usuario con diagramas de transición entre interfaces de usuario

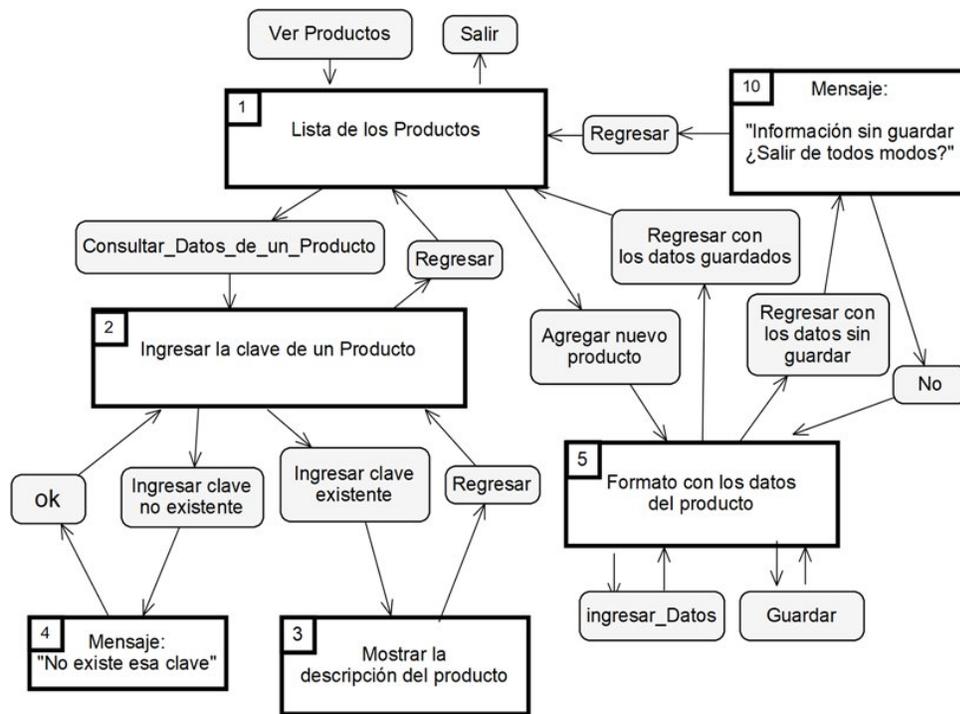
La descripción de las transiciones entre las interfaces de usuario juega un papel importante en el desarrollo de un sistema de software a gran escala, ya que ayuda a los desarrolladores a tener más claro el funcionamiento no solamente de la parte que les toca desarrollar, sino también del funcionamiento de todo el sistema.

La especificación de las interacciones entre el sistema y los usuarios, y las interfaces que tendrá un sistema de software son cruciales durante la especificación de requerimientos de un producto de software, ya que debe quedar claro cuáles son exactamente los servicios que se pueden solicitar al sistema. Los Diagramas de Transiciones entre Interfaces de Usuario (DTIU) [15] son una notación de modelado que permite modelar sistemas de software de forma clara e intuitiva. La notación DTIU se diseñó para simplificar la especificación y el diseño de las interacciones entre el sistema y el usuario sin perder los detalles técnicos necesarios para desarrollar el sistema.

Un DTIU es un grafo dirigido con nodos que representan interfaces de usuario y aristas que representan transiciones entre interfaces de usuario. Cada transición tiene una etiqueta. Las etiquetas en las transiciones se componen de una acción del usuario y, cuando dos transiciones tienen la misma acción del usuario, la etiqueta también contiene información sobre las condiciones que deben cumplirse para activar la transición. Las transiciones tienen una interfaz de usuario de origen y una de destino.

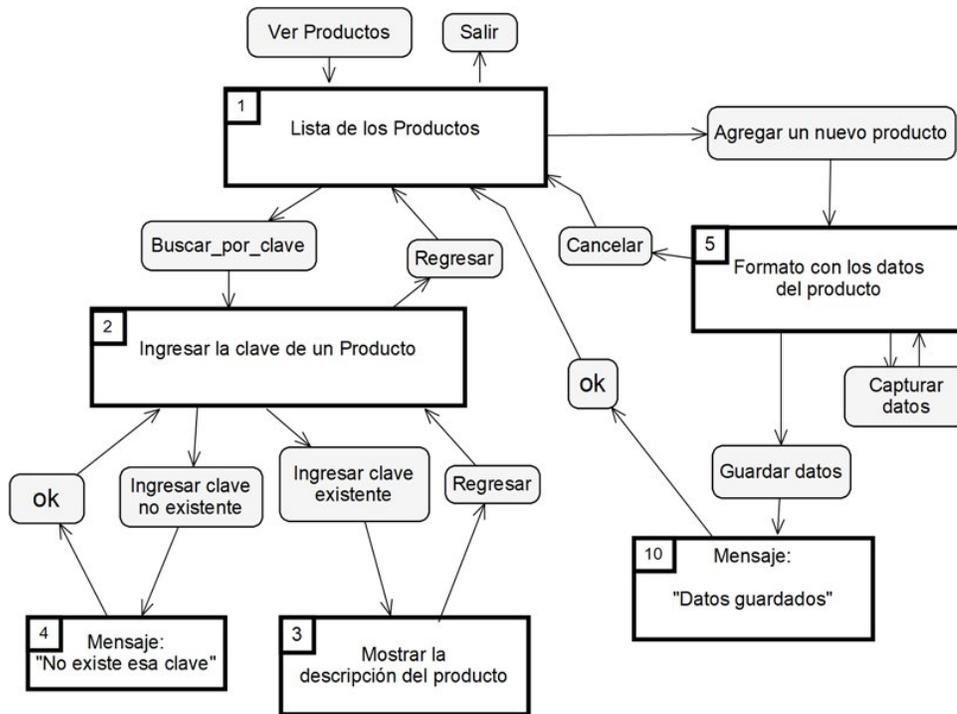
Un DTIU contiene el nombre de cada una de las interfaces de usuario y un número que permite una fácil identificación durante el diseño y las etapas posteriores.

La Figura 5.29 ilustra un ejemplo de un DTIU que describe parcialmente un sistema. En éste, la interfaz #1 (Lista de los Productos) se despliega cuando el usuario solicita *ver productos*. La interfaz #1 tiene tres transiciones posibles: i) *Buscar\_por\_clave* hace que se despliegue la interfaz #2 (Ingresar la clave de un producto), ii) *Agregar un nuevo producto* y iii) *Salir*. Las interfaces #2 y #5 también tienen sus respectivas transiciones y cada una de ellas conduce a su interfaz de destino correspondiente. En la interfaz #2, cuando el usuario ingresa la clave de un producto, se pueden activar dos transiciones: “Ingresar clave existente” e “Ingresar clave no existente”. Por lo tanto, la transición que se activará depende de la condición: *clave existente* o *clave no existente*.



**Figura 5.29** Ejemplo de un DTIU que describe una parte de un sistema.

Los DTIU se pueden ir rediseñando hasta encontrar un buen diseño. Por ejemplo, en el DTIU de la Figura 5.29, el usuario debe seleccionar el botón “Guardar” y después el botón “Regresar” para que sus datos queden guardados. En este caso, existe el riesgo de que regrese sin haber guardado los datos y habrá que advertirle mediante el mensaje de la interfaz # 10. El DTIU de la Figura 5.30 ilustra un diseño mejorado. Como se puede apreciar en este caso, cuando el usuario selecciona el botón “Guardar”, el sistema regresa automáticamente a la interfaz de usuario anterior. De esta manera, en lugar de la opción “Regresar” existe la opción “Cancelar”, la cual indica claramente que se está cancelando la captura de los datos.



**Figura 5.30** Ejemplo de un DTIU en el que se hace un rediseño.

Con los DTIU se puede hacer referencia cruzada de elementos entre los requerimientos y el diseño. El lector puede consultar ejemplos de diseño en el que se utilizan los DTIU en [16-18].

## V.7 Diagrama de Secuencias Detallado en el Diseño Detallado

Los Diagramas de Secuencia Detallados (DSD) [17] son una particularización de los diagramas de secuencia de UML, que tienen como objetivo ayudar a cumplir tres objetivos de diseño fundamentales:

- i) Anticipar y resolver posibles problemas,
- ii) Especificar los requerimientos de cada módulo en la arquitectura del software
- iii) Garantizar que se cumplirán los requisitos del sistema si la implementación sigue el diseño.

Para lograr estos objetivos, los DSD tienen las siguientes reglas generales para los diagramas de secuencia:

1. Cada secuencia y subsecuencia debe identificarse con una identidad y un nombre únicos. Las secuencias que comienzan con una acción del usuario deben tener una identidad que comience con un número igual al número de la Interfaz de Usuario (IU) del DTIU en el que se realiza la acción; de lo contrario, la identidad de la secuencia debe comenzar con una letra.

2. Todos los módulos que participan en una acción del usuario o del sistema deben mostrarse en una columna separada.

3. Debe haber una declaración "ir a" al final de cada secuencia/subsecuencia con una referencia a todas las secuencias de continuación posibles.

4. Las funciones que deben realizarse internamente dentro de un módulo deben indicarse en un cuadro de texto usando lenguaje natural.

5. Los mensajes de un módulo a otro deben tener una etiqueta para indicar la identidad del mensaje y los parámetros que "viajan" en él. En caso de una llamada a un método, la identidad del mensaje es el nombre del método. Los mensajes devueltos de una llamada al método no tienen identidad (ver la regla 6).

6. Se utiliza una flecha con línea punteada para indicar un mensaje de retorno (de un mensaje de llamada de método). En este tipo de mensaje, se indican los tipos de datos devueltos (ver la regla 7).

7. En caso de que la secuencia dependa del valor específico enviado en un parámetro de mensaje, la etiqueta del mensaje debe incluir este valor específico. Se debe hacer una secuencia separada para cada caso iniciado por el mensaje que contiene dicho valor.

En la siguiente sección describimos un estudio de caso sobre un sistema de gestión de viajes compartidos junto con ejemplos ilustrativos de cómo utilizar el DTIU y el DSD en el diseño.

Con el DTIU se describe completamente el comportamiento de las interfaces de usuario del sistema, y con DSD se especifica claramente la secuencia de mensajes entre los módulos del sistema. Con el DSD se especifican también las tareas requeridas de los módulos y el momento en que deben ejecutarse.

En la siguiente sección se explica la forma en la que los DTIU combinados con los DSD pueden ser de mucha utilidad para el diseño.

## V.8 Caso de estudio 1

### V.8.1 DTIU y DSD en el diseño de un sistema gestor de uso compartido de vehículos

Viaja Sin Tránsito (VST) es un sistema gratuito que funciona en red, y pretende ayudar a compartir automóviles entre los alumnos de la UAM Cuajimalpa. Está concebido para que brinde un servicio confiable para los usuarios, tanto pasajeros como conductores. Garantizando, en la medida de lo posible, que tanto los conductores como los pasajeros sean miembros de la comunidad. Este proyecto es una adaptación de la idea "The Waze Car Pool" para la comunidad de la UAM-C. El sistema tiene sentido cuando hay conductores que desean compartir los gastos

con los pasajeros. Cuando un usuario de VST desea compartir su automóvil, crea un itinerario con los datos de su ruta y luego adquiere el rol de Conductor del itinerario creado. Cuando un usuario solicita una reserva para un itinerario, él / ella adquiere el rol de Pasajero de ese itinerario. Cada usuario podrá ver la información de acuerdo con su rol.

### Características del usuario

Los conductores y los usuarios de VST deben pertenecer a la unidad UAM-C. Para garantizar lo anterior, se solicita a los usuarios que suban al sistema una fotografía de la Credencial UAM-C y otra de su última tira de materias.

### Limitaciones generales

- La aplicación no maneja transacciones económicas.
- Cada usuario podrá ver la información que le corresponde de acuerdo con su rol.

### Descripción del funcionamiento

Las acciones que se pueden realizar con los itinerarios dependen de las siguientes condiciones:

1) *El itinerario está activo o suspendido.* - Los usuarios pueden seleccionar todos los itinerarios que están en estado "Activo", pero si un itinerario está en estado "Suspendido", solo el propietario puede seleccionarlo.

2) *El usuario es el conductor del itinerario.* - Solo el usuario que creó el itinerario (el conductor) puede realizar cambios en él.

3) *El usuario no es conductor ni pasajero.* - En este caso, el usuario puede solicitar una reserva en el itinerario que seleccionó.

4) *El usuario es un pasajero del itinerario.* - Cuando el usuario activo tiene una reserva registrada o tiene una solicitud de reserva en un itinerario, se convierte en Pasajero. El Pasajero de un itinerario puede solicitar la cancelación de su reserva.

La interfaz de usuario que se muestra al seleccionar uno de los itinerarios en la interfaz de Inicio depende de las condiciones mencionadas anteriormente. Las acciones que el usuario puede realizar se especifican a continuación para cada uno de los cuatro casos posibles.

1. *Seleccionar un itinerario suspendido siendo el conductor:* En este caso, el conductor puede reactivar su itinerario.
2. *Seleccionar un itinerario activo siendo el conductor:* En este caso, el controlador puede realizar cualquiera de las siguientes acciones.
  - a) Modificar datos del itinerario.
  - b) Suspende el itinerario.

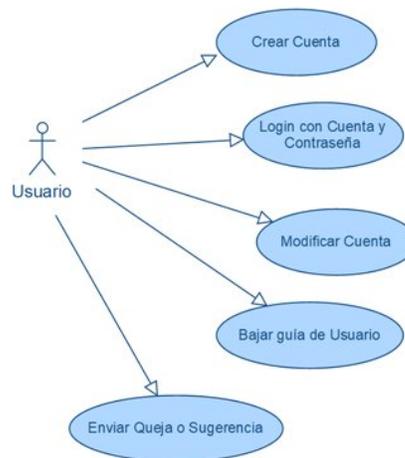
- c) Aceptar o rechazar una solicitud de reserva.
- d) Cancelar una reserva registrada.
- e) Calificar a un pasajero.

3. *Seleccionar un itinerario activo sin ser el conductor y sin tener una reserva:* Cualquier usuario puede consultar los itinerarios disponibles. Cuando él / ella selecciona uno, él / ella puede solicitar una reserva, en este caso el sistema envía un correo electrónico al conductor notificándole la solicitud. Además, envía todos los datos del usuario candidato para ser su pasajero.

4. *Seleccionar un itinerario activo sin ser el conductor teniendo una reserva:* En este caso, el pasajero puede solicitar la cancelación de su reserva. El sistema envía un correo electrónico al conductor notificando la solicitud de cancelación de la reserva del pasajero.

### Los casos de uso del rol de usuario

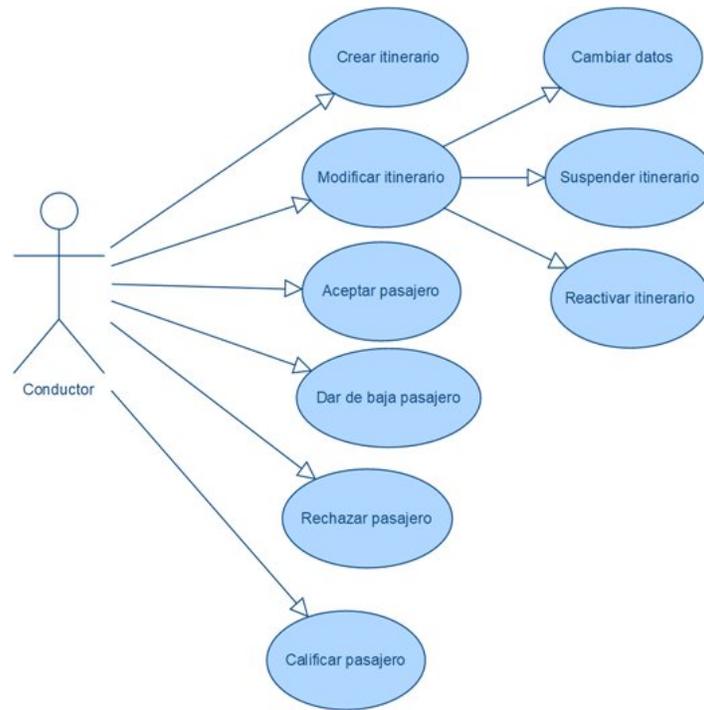
Para poder acceder al sistema es necesario darse de alta como usuario. En la Figura 5.31 se ilustran los casos de uso para el rol de Usuario de VST.



**Figura 5.31** Casos de Uso para el rol de *Usuario* de VST

### Los casos de uso del rol de conductor

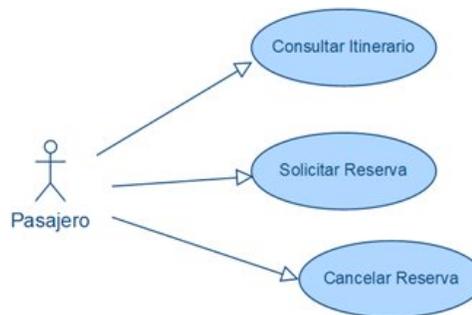
Cualquier usuario podrá tener el rol de conductor cuando crea un Itinerario. En la Figura 5.32 se ilustran los casos de uso del conductor.



**Figura 5.32** Casos de Uso para el rol de *Conductor* de VST

### Los casos de uso del rol de pasajero

Cualquier usuario podrá tener el rol de pasajero cuando se inscribe a un Itinerario. En la Figura 5.33 se ilustran los casos de uso del pasajero.



**Figura 5.33** Casos de Uso para el rol de *Pasajero* de VST

En el diseño de VST las transiciones entre interfaces de usuario se modelaron con DTIU. En la Figura 5.34 se ilustra el DTIU de la interfaz principal (home) de VST.

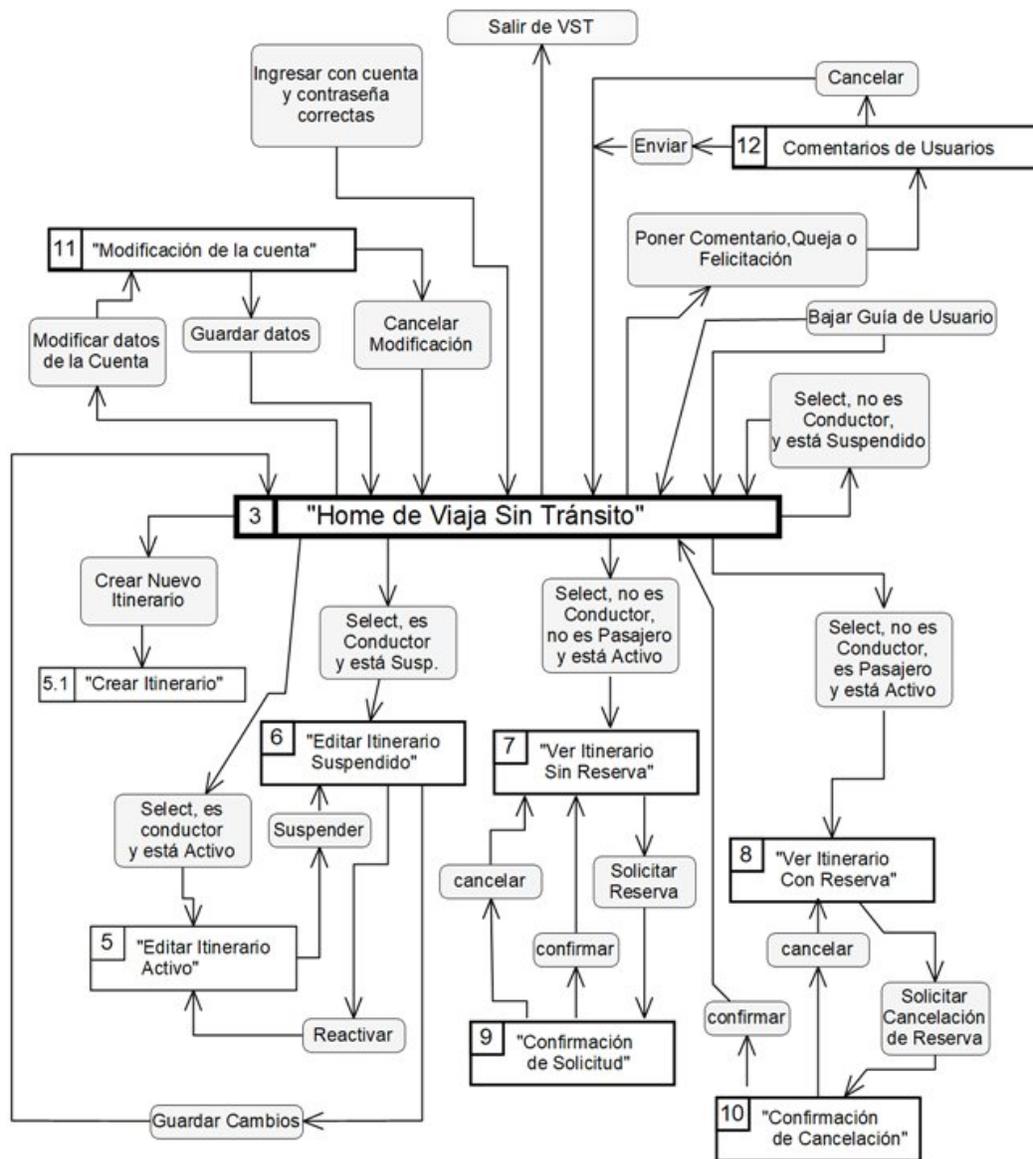
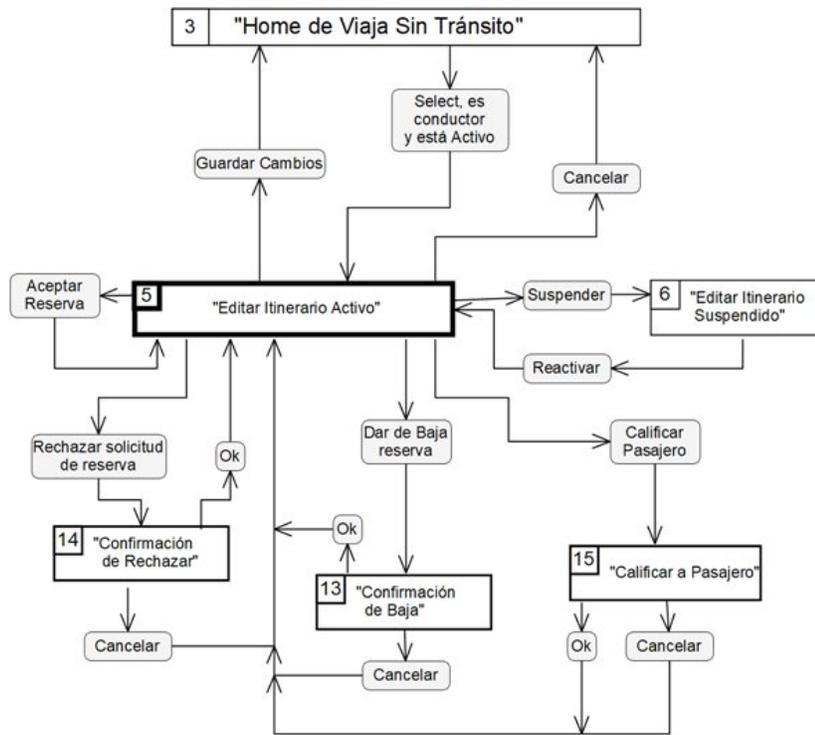


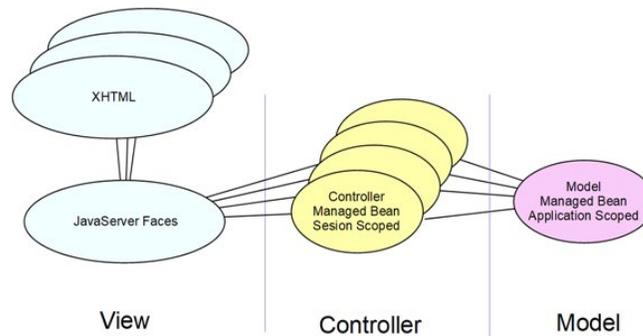
Figura 5.34 DTIU del "Home de VST"

Las interfaces de usuario cuyo contorno está en negrita en un DTIU, como es el caso de la interfaz #3 del DTIU de la Figura 5.34, contienen todas las acciones de usuario posibles en dicha interfaz. Para que los diagramas sean más claros, el conjunto completo de acciones de usuario de una interfaz, se puede detallar en un DTIU por separado. Por ejemplo, en la Figura 5.35 se muestra el DTIU de la interfaz #5 "Editar Itinerario Activo".



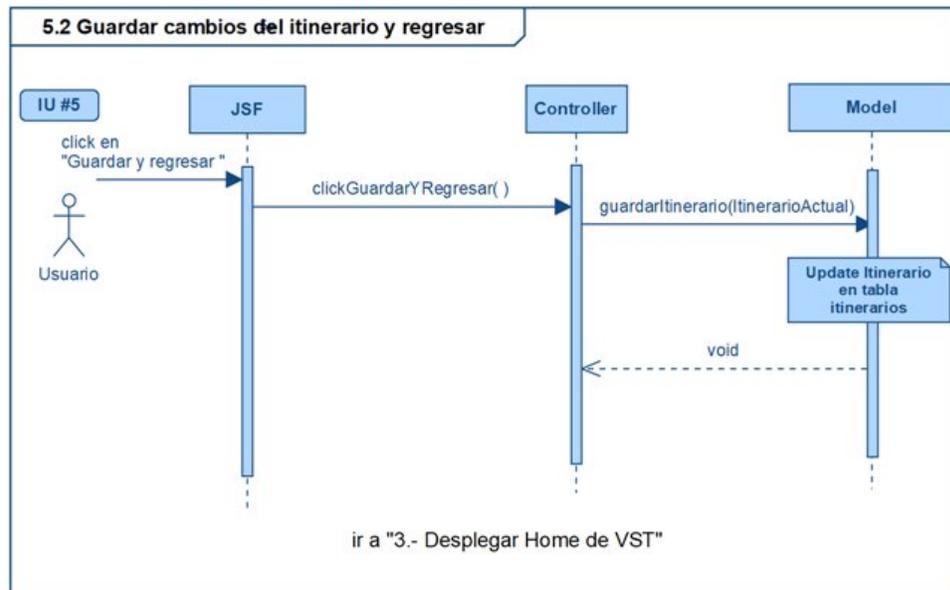
**Figura 5.35** DTIU del "Home de VST"

La Figura 5.36 muestra la arquitectura del sistema. En este caso, tenemos una aplicación web implementada con la tecnología JavaServer Faces [19], que se basa en beans gestionados. Un bean gestionado es un módulo Java al que se puede acceder desde una página web (archivo xhtml) llamando a sus métodos. Se puede ver en la Figura 5.36 que el sistema se basa en el patrón Modelo-Vista-Controlador. El modelo es una clase de instancia única y siempre está activa en la aplicación, por lo que es un bean de ámbito (alcance) de aplicación. Hay una instancia del controlador que está activo durante cada sesión de usuario individual, por lo que es un bean de sesión. Las páginas xhtml son administradas por el sistema JSF que a su vez se comunica con el controlador cuando una página xhtml lo indica.



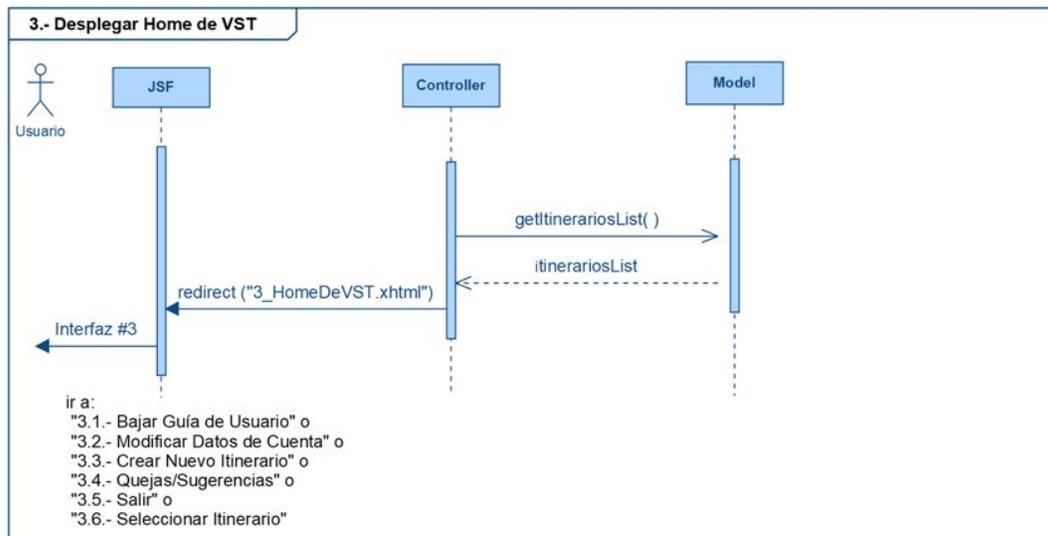
**Figura 5.36** Arquitectura de VST

La Figura 5.37 ejemplifica algunos aspectos de las reglas de los DSD. Este caso particular el diagrama de secuencia ilustra lo que sucede cuando el usuario selecciona el botón "Guardar y volver" en la interfaz # 5: "Editar itinerario activo". El número de secuencia 5.2 indica la identidad de la interfaz en la que tiene lugar la acción del usuario (# 5) seguido de un número (2) que identifica una acción del usuario dentro de la IU. El nombre de la secuencia describe la acción del usuario (Guardar cambios y volver) que la inicia.



**Figura 5.37** DSD de "Guardar cambios del Itinerario y Regresar"

El servidor que ejecuta JavaServer Faces se representa como el módulo JSF. Entonces, los módulos que participan en esta acción son: JSF, el Controller y el Model. Cuando el usuario hace clic en el botón "Guardar y regresar", JSF llama al método de escucha indicado en el archivo xhtml, en este caso: `clickGuardarYRegresar()` que se encuentra en el Controller. En este método, el Controller ordena al Model que almacene el itinerario editado invocando el método: `guardarItinerario()`, que recibe como parámetro el itinerario actual. En este método, el Model es responsable de actualizar la tabla "Itinerarios" en la base de datos con los datos del itinerario recibido. La flecha con línea punteada que va del Model al Controller indica que el Model devuelve el control al Controller con un valor de retorno nulo. Finalmente, la instrucción "ir a" al final de la secuencia indica que la secuencia continúa con la subsecuencia "3.- Desplegar Home de VST", tal como se muestra en la Figura 5.38.



**Figura 5.38** DSD de “Desplegar Home de VST”

En la Figura 5.38, el Controller solicita al Model la lista actualizada de itinerarios y, cuando ésta llega, ordena la transición a la interfaz # 3. La información en la parte inferior del diagrama indica que la continuación de la secuencia dependerá de la próxima acción del usuario dentro de la interfaz # 3. Nótese que estas acciones son las mismas que se indican en las etiquetas de transición en el DTIU de la Figura 5.35. A continuación se muestran las reglas para documentar aspectos específicos en el DSD.

### Reglas para documentar aspectos específicos en el DSD

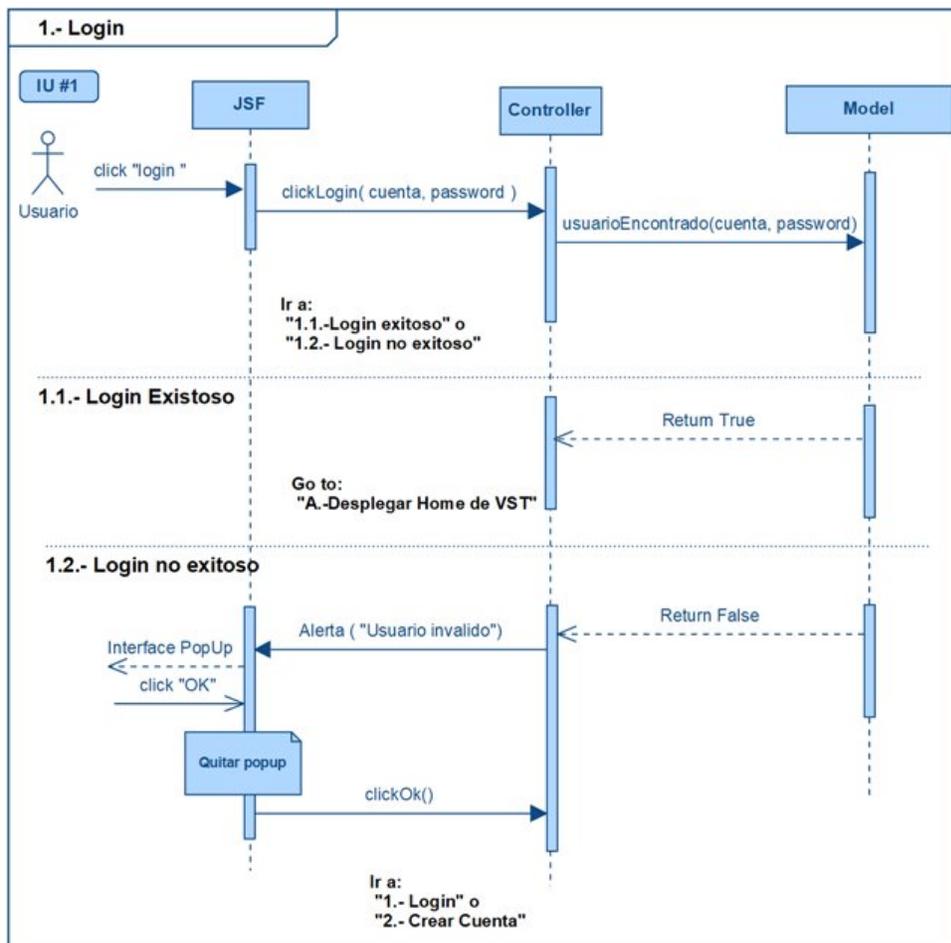
En el epígrafe V.4 establecimos reglas generales para la construcción de un DSD. A continuación explicamos las reglas para documentar en un DSD dos aspectos específicos: casos booleanos y especificación de varias tareas clave en un módulo.

*Documentación de casos booleanos:* Los casos booleanos son aquellos en los que se envía un mensaje de un módulo a otro esperando recibir más tarde un valor booleano como resultado. Estas secuencias se documentan de la siguiente manera:

- Cuando se invoca un método que devuelve un valor booleano, se coloca una instrucción go-to que indica el nombre de las dos posibles secuencias que siguen a continuación.
- Se utiliza una línea horizontal punteada para separar la secuencia actual de las dos secuencias posibles y cada uno de estos casos se identifica con una etiqueta.
- Una de las secuencias de continuación debe comenzar con un valor de retorno verdadero y la otra con un valor de retorno falso (flecha punteada).

Por ejemplo, la Figura 5.39 muestra cómo indicar esto en un DSD. Cuando el método `usuarioEncontrado()` devuelve *verdadero*, significa que la cuenta y la

contraseña proporcionadas por el usuario se encontraron en la base de datos, entonces el control se transfiere a la secuencia "1.1 Login exitoso". De lo contrario, el control se transfiere a la secuencia "1.2 Login no exitoso", en el que se muestra una ventana emergente (PopUp) con una notificación para el usuario.



**Figura 5.39** DSD de la acción "Inicio de Sesión" (login)

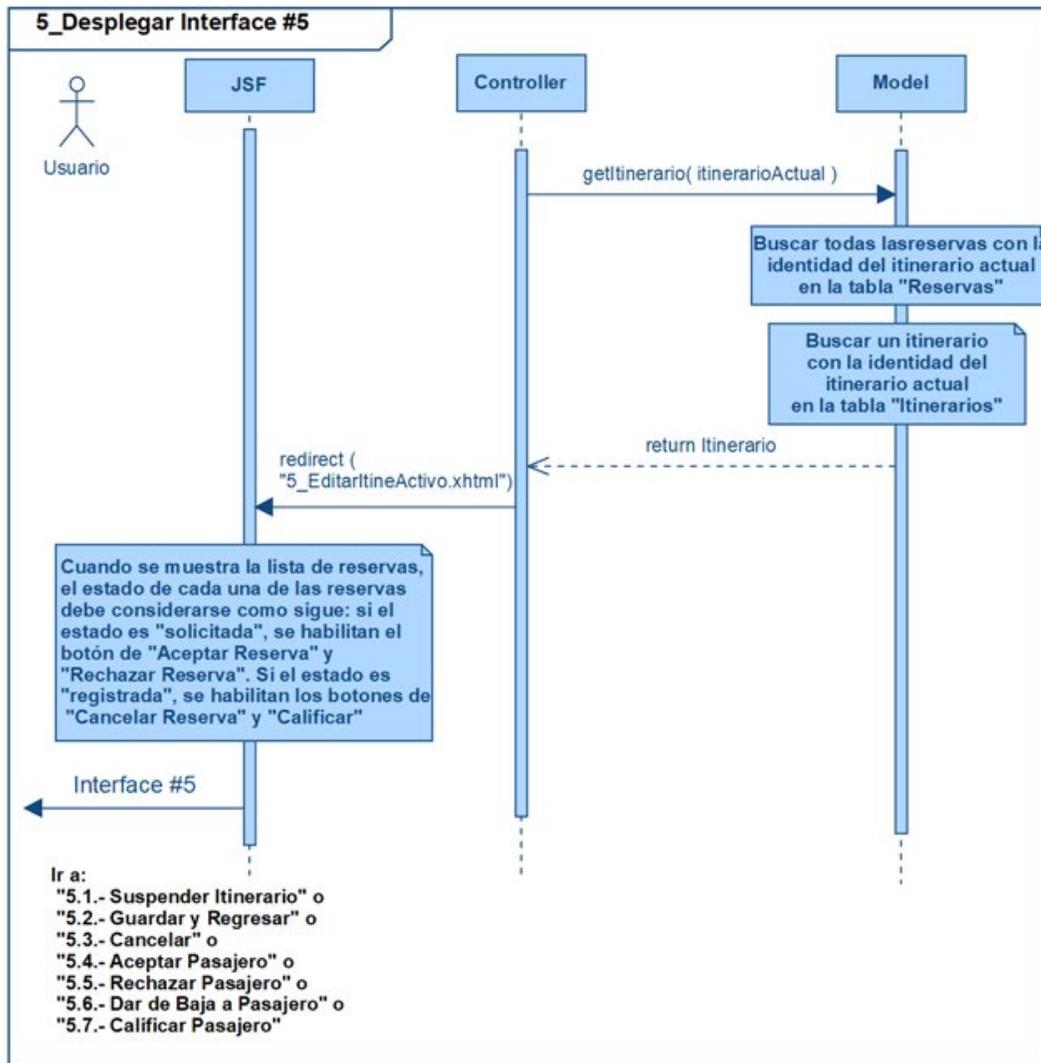
**Especificación de varias tareas clave en un módulo.**

Una de las ventajas del DSD es que el diseñador puede documentar en detalle las tareas realizadas por un módulo en cada momento sin incluir los detalles de implementación. Esto es muy útil cuando los programadores que hacen la codificación son novatos, porque el riesgo de fallas causadas por la omisión de tareas disminuye.

*Regla para especificar varias tareas clave en un módulo:*

- Se debe utilizar lenguaje natural para especificar los requerimientos de las tareas internas de un módulo.

La Figura 5.40 es un ejemplo de la documentación detallada de las tareas que se ejecutan en el Model y en JSF cuando se muestra la interfaz # 5.



**Figura 5.40** DSD de la acción "Inicio de Sesión" (login)

Es muy importante que el diseño de los proyectos de software a gran escala esté lo suficientemente bien documentado, pues esto ayuda a lidiar con la complejidad. Si el diseño no contiene la información suficiente, se dificultará llevar a cabo la corrección de defectos encontrados en las pruebas y el mantenimiento.

Los Diagramas de Transición entre Interfaces de Usuario (DTIU) combinados con los Diagramas de Secuencia Detallados (DSD) son muy útiles para ilustrar las decisiones de diseño. El DSD ayuda a lograr un diseño lo suficientemente abstracto para ocultar los detalles innecesarios, pero al mismo tiempo lo suficientemente específico como para proporcionar una especificación completa de los requerimientos de cada uno de los módulos de software.

Lo anterior, además de facilitar el mantenimiento, promueve la disminución de los errores debidos a la falta de información durante la implementación (por ejemplo, no acceder a la base de datos antes de mostrar una interfaz). El DSD combinado con el DTIU permiten definir, de manera completa y precisa para los programadores, la manera en la que se presenta cada interfaz de usuario según ciertas condiciones (por ejemplo, si el itinerario está activo o no, si el usuario es conductor, pasajero o ninguno). Además, ayudan a explicar fácilmente a los programadores cómo y cuándo se comunican entre sí los módulos del sistema.

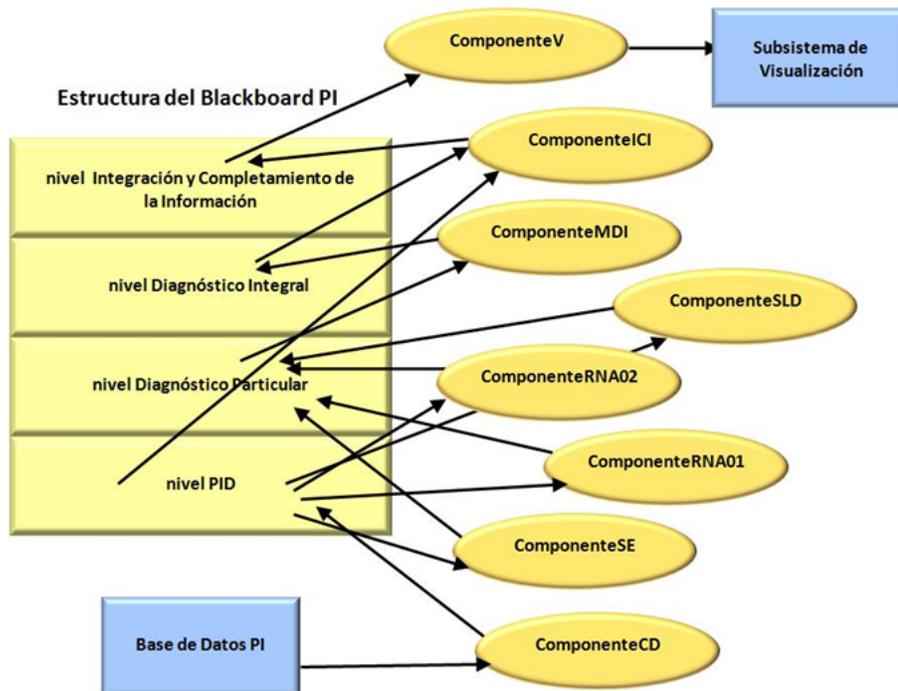
## V.9 Casos de estudio 2

En este epígrafe retomaremos los dos casos de estudio, a los cuales nos hemos referido en los capítulos previos, e ilustraremos como en ambos casos la solución arquitectónica consistió en la combinación de la arquitectura de pizarra con la arquitectura de sistemas interactivos MVC. En ambos casos, la arquitectura de pizarra fue utilizada para modelar los componentes de la lógica del sistema. Ambos enfoques arquitectónicos serán expuestos utilizando los modelos de vistas introducidos previamente.

### V.9.1 Sistema de detección de fugas en ductos que transportan hidrocarburos. Diseño arquitectónico

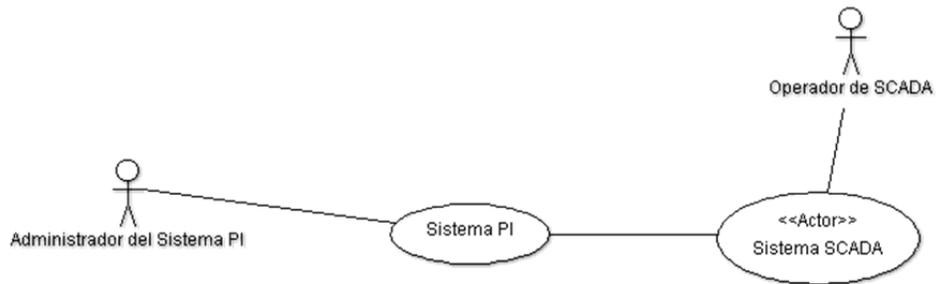
Como mencionamos previamente, el diseño arquitectónico del subsistema SPI se basa en la integración de los modelos arquitectura de pizarra y arquitectura MVC en una misma solución arquitectónica. Para el diseño de la arquitectura del subsistema SPI se utilizó el Modelo de Vistas 4+1.

En la Figura 5.41 se muestra como la arquitectura de pizarra fue utilizada para modelar los componentes de la lógica del subsistema SPI, al considerar el procesamiento integral de los eventos de fugas notificados, como un proceso de solución oportunista e incremental. Como se puede apreciar en esta figura, el pizarrón (denotado como Blackboard PI) ha sido estructurado en cuatro niveles – nivel PID, nivel Diagnóstico Particular, nivel Diagnóstico Integral y nivel Integración y Completamiento de la Información– donde el nivel “PID” representa el nivel de mayor abstracción, mientras que el nivel “Integración y Completamiento de la Información” representa el nivel de mayor detalle o elaboración de la solución. Por otra parte, las fuentes de conocimiento modelan los componentes envueltos en el procesamiento integral (diagnósticos particulares y diagnóstico integral) del PID recibido en el primer nivel del pizarrón. Ejemplos de fuentes de conocimiento son los componentes *ComponenteSE*, *ComponenteRNA01*, *ComponenteRNA02*, *ComponenteSLD* y *ComponenteMDI*.

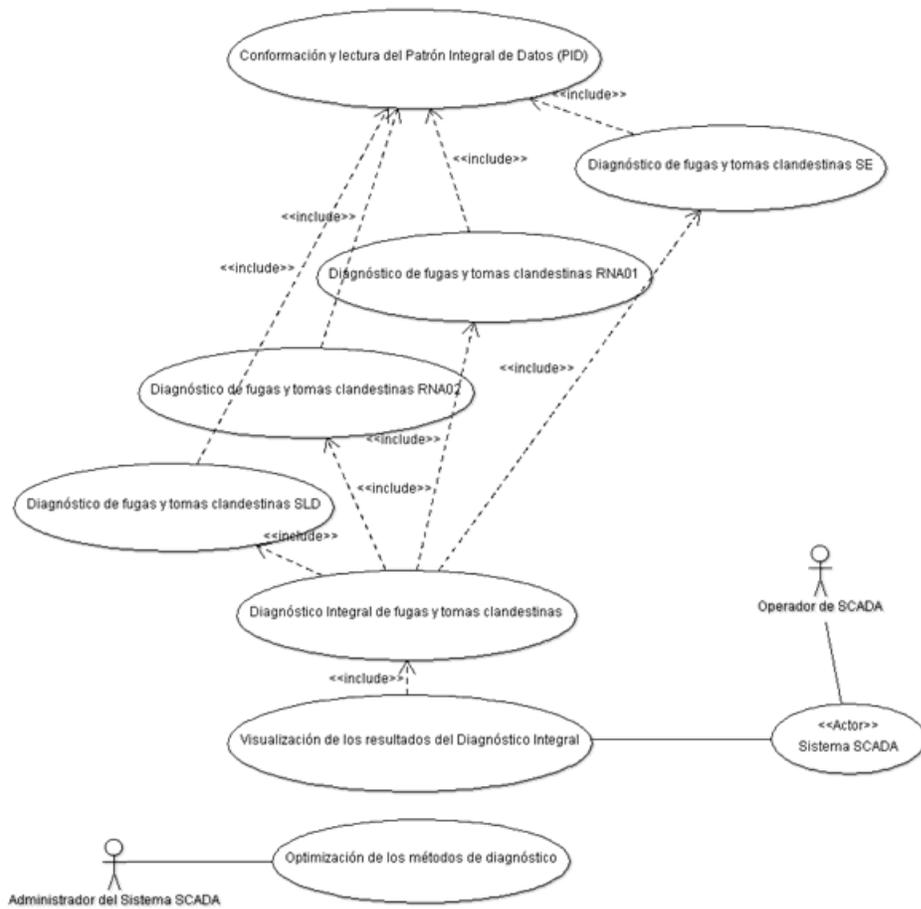


**Figura 5.41** La arquitectura pizarra en el diseño e implementación del subsistema SPI. Nótese que el procesamiento integral de los eventos de fugas notificados al subsistema SPI, se ejecuta de una forma incremental, a través de los cuatro niveles de abstracción que conforman la pizarra. El nivel “PID” representa el nivel de mayor abstracción, mientras que el nivel “Integración y Completamiento de la Información” representa el nivel de mayor detalle o elaboración de la solución.

Las Figuras 5.42 y 5.43 ilustran parte de la Vista de Casos de Uso. Aspectos de la Vista Lógica son ilustrados en las Figuras 5.44 y 5.45, abarcando parte del diseño orientado a objetos propuesto para la arquitectura de pizarra mostrada en la Figura 5.41. Nótese que este diseño orientado a objetos corresponde a la lógica de la aplicación, es decir, a la capa Modelo, considerando una arquitectura MVC que engloba a la arquitectura de pizarra. Las Figuras 5.46 y 5.47 exhiben aspectos de la Vista de Proceso. En la Figura 5.48 se muestra un aspecto de la Vista de Desarrollo. Finalmente, en la Figura 5.49 se puede apreciar una de las propuestas de Vista Física del sistema SPI.



**Figura 5.42** La Vista de Casos de Uso del sistema SPI. Detalle A.



**Figura 5.43** La Vista de Casos de Uso del sistema SPI. Detalle B.

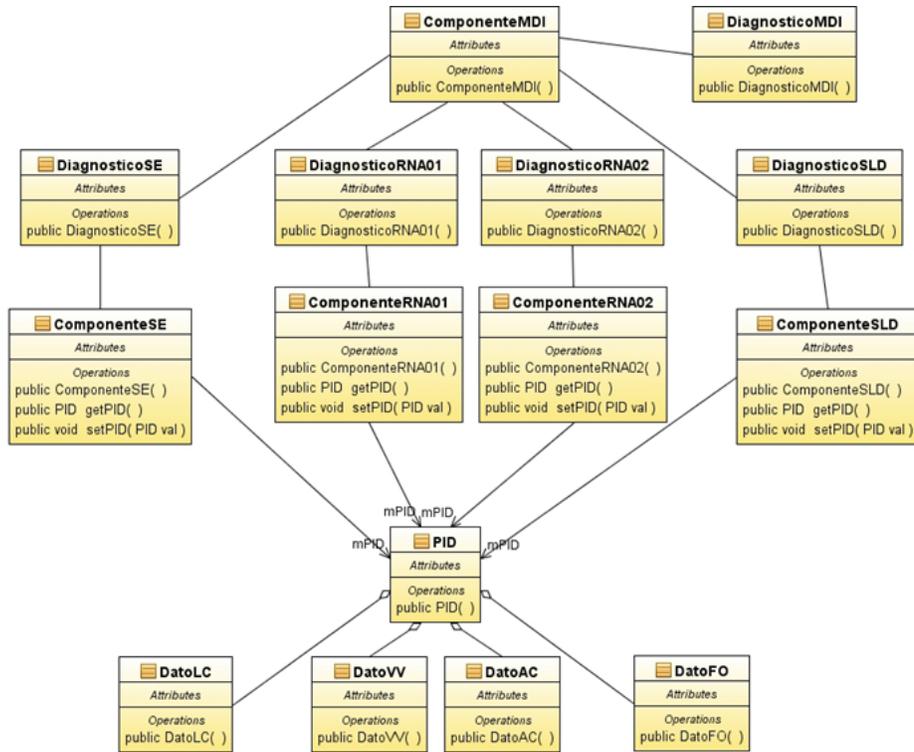


Figura 5.44 La Vista Lógica del sistema SPI. Detalle A.

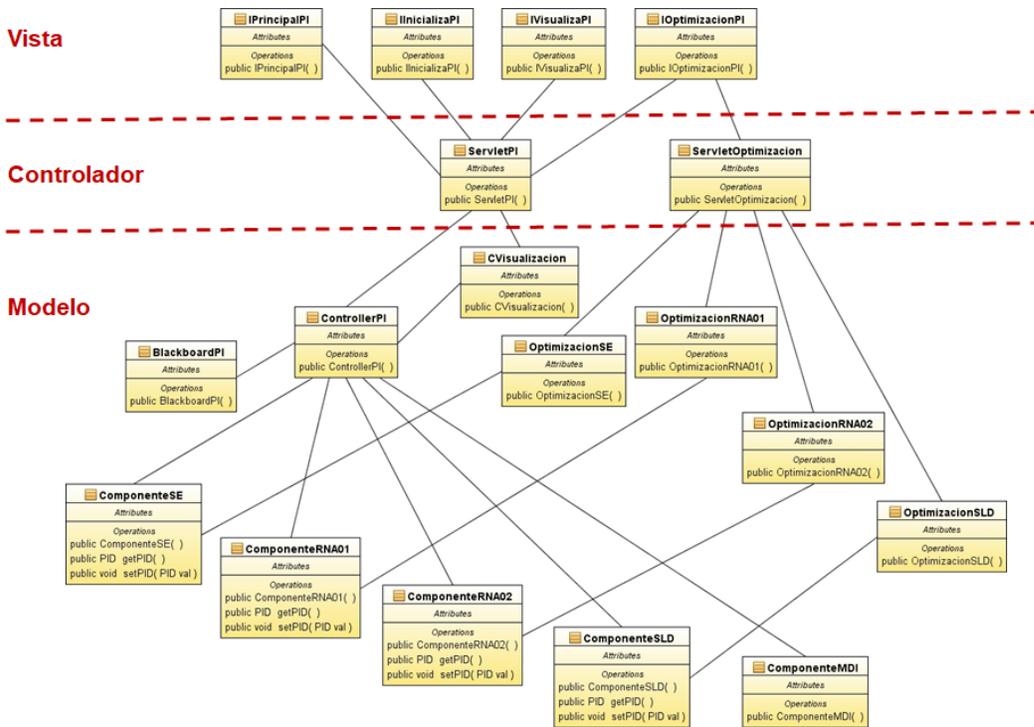


Figura 5.45 La Vista Lógica del sistema SPI. Detalle B.

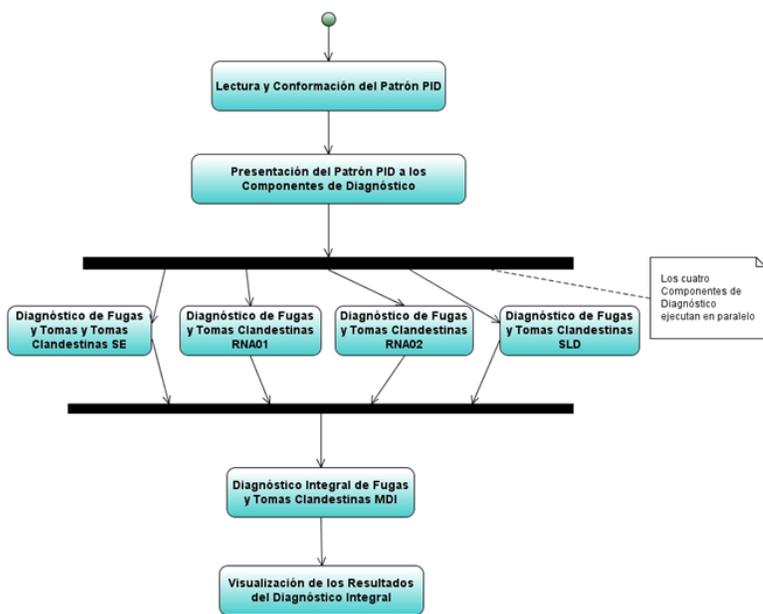


Figura 5.46 La Vista de Proceso del sistema SPI. Detalle A.

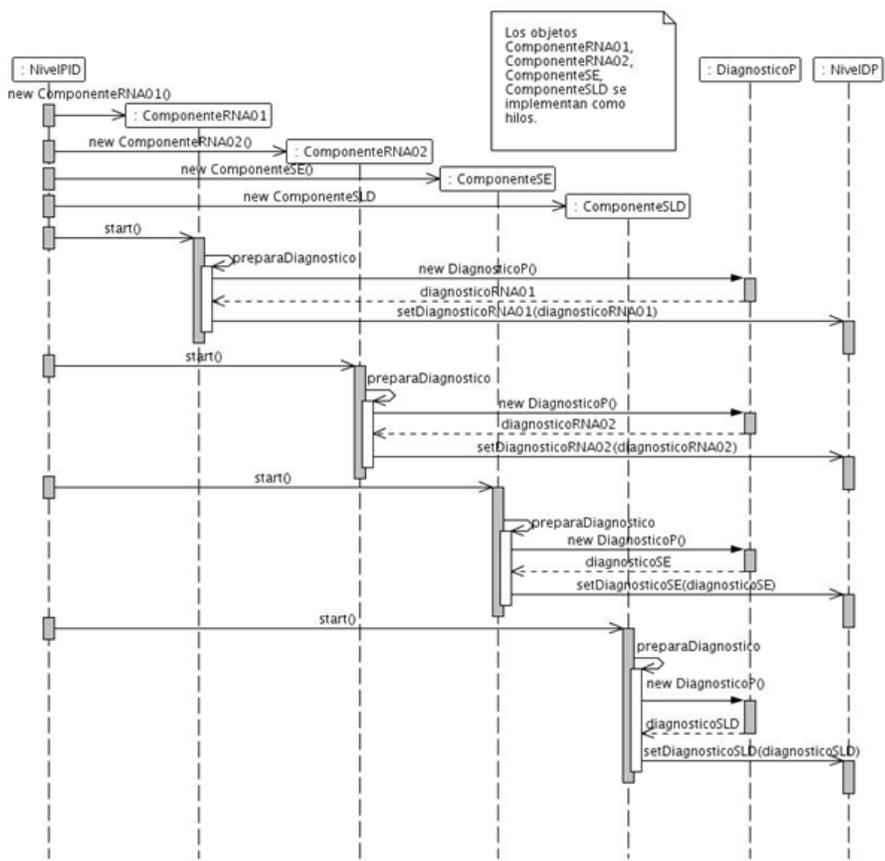
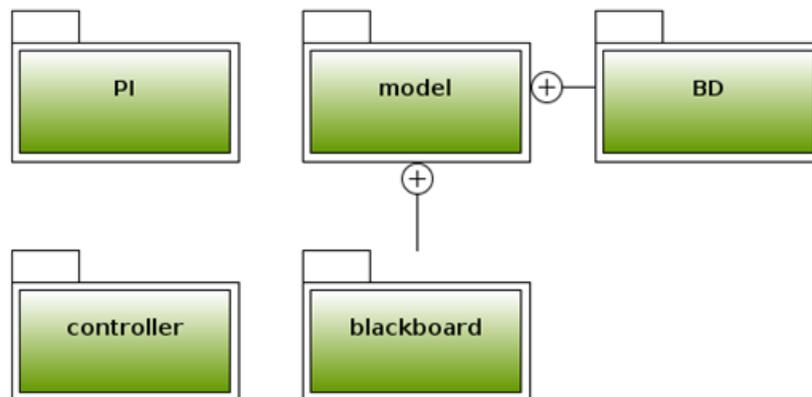
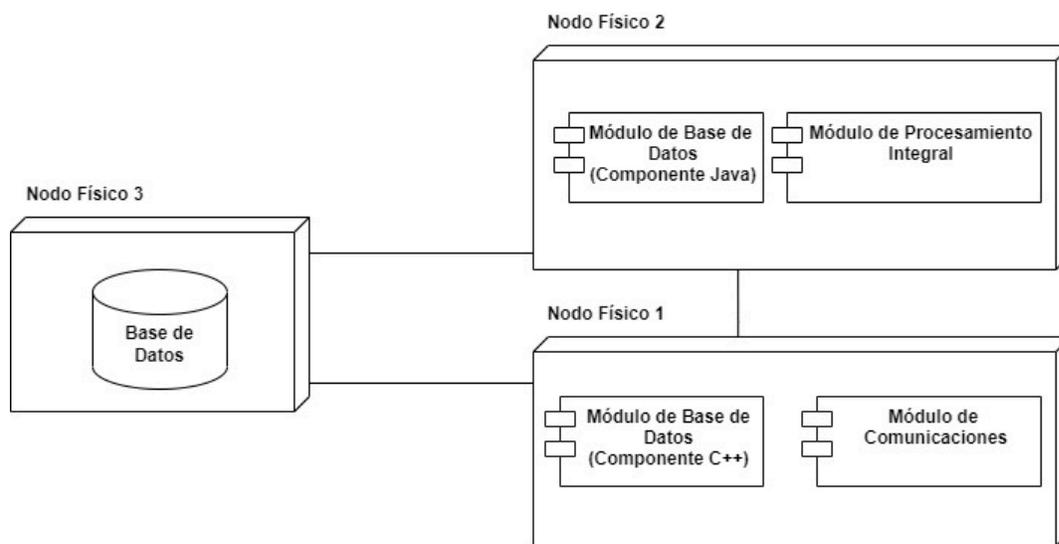


Figura 5.47 La Vista de Proceso del sistema SPI. Detalle B.

Diseño Arquitectónico y Diseño de la Interacción entre Interfaces Gráficas de Usuario — Casos de estudio 2



**Figura 5.48** La Vista de Desarrollo del sistema SPI. Detalle.

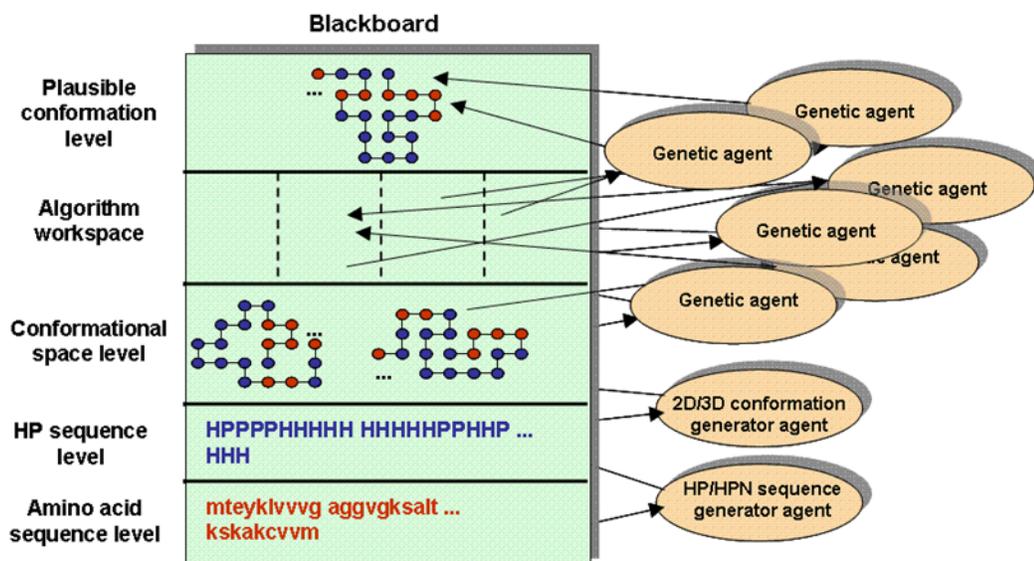


**Figura 5.49** La Vista Física del sistema SPI. Detalle.

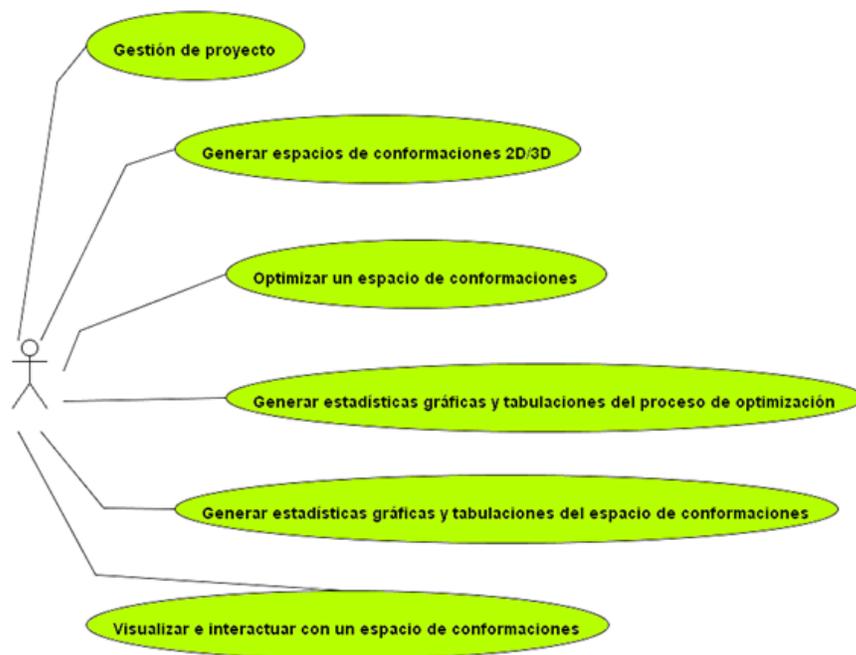
## V.9.2 Plataforma Bioinformática Evolution. Diseño Arquitectónico

Como ya habíamos mencionado, la plataforma bioinformática Evolution es otro ejemplo de sistema de software que integra las arquitecturas MVC y pizarra en una misma solución arquitectónica. Para el diseño de la arquitectura de la plataforma Evolution se utilizó el Modelo de Vistas del Proceso Unificado. La Figura 5.50 muestra el uso de la arquitectura de pizarra en la plataforma bioinformática Evolution. La Figura 5.51 muestra parte de la Vista de Caso de Usos. Aspectos de la Vista Lógica son ilustrados en las Figuras 5.52 a la 5.55. Con relación a la Vista de Procesos, algunos de los aspectos capturados por la misma se ilustran en las

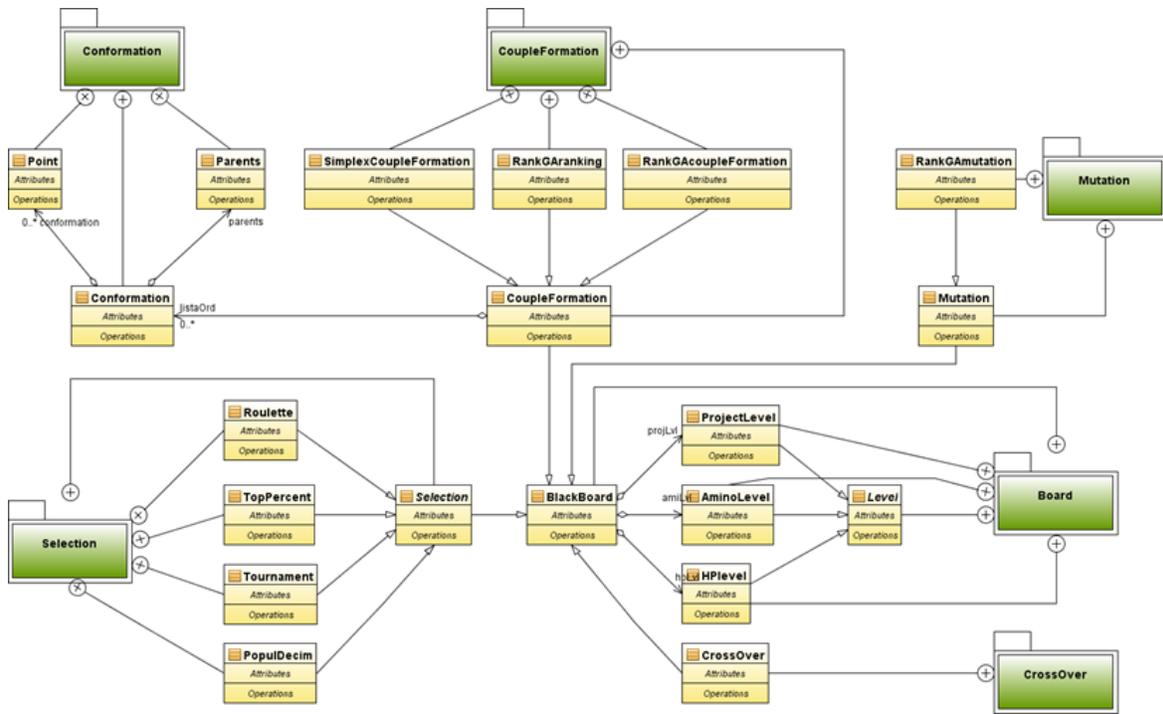
Figuras 5.56 y 5.57. Parte del modelo de datos propuesto se puede apreciar en la Vista de Datos que se ilustra en las Figuras 5.58 y 5.59. Finalmente, las Figuras 5.60 y 5.61 exhiben algunos aspectos de la Vista de Despliegue.



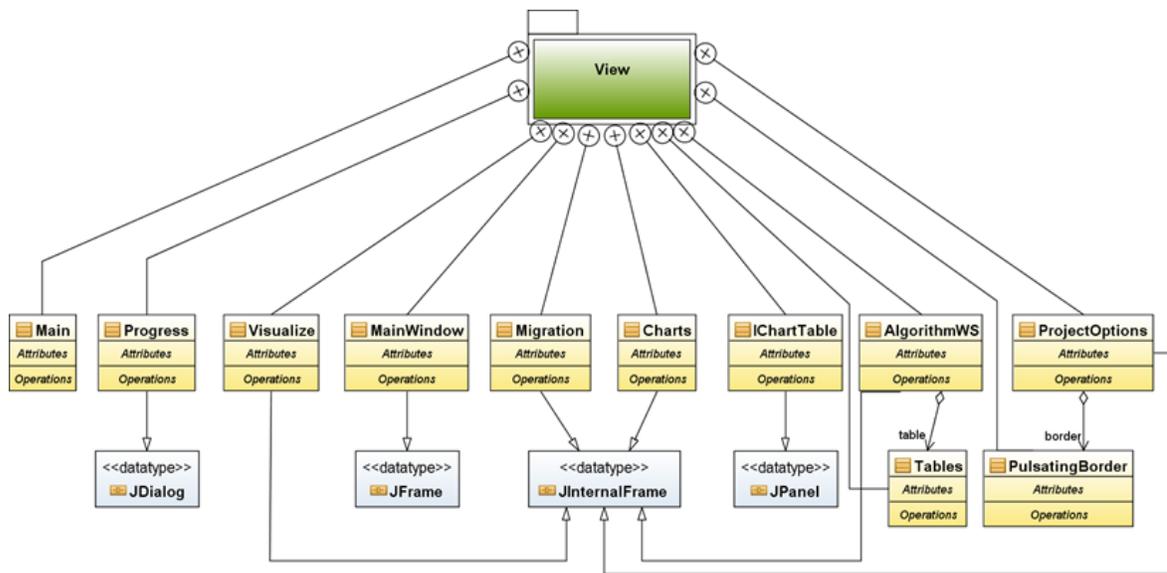
**Figura 5.50** Modelo de arquitectura de pizarra de la plataforma bioinformática Evolution.



**Figura 5.51** Vista de Casos de Uso de la plataforma bioinformática Evolution. Detalle.



**Figura 5.52** Vista Lógica de la plataforma bioinformática Evolution. Detalle A.



**Figura 5.53** Vista Lógica de la plataforma bioinformática Evolution. Detalle B.

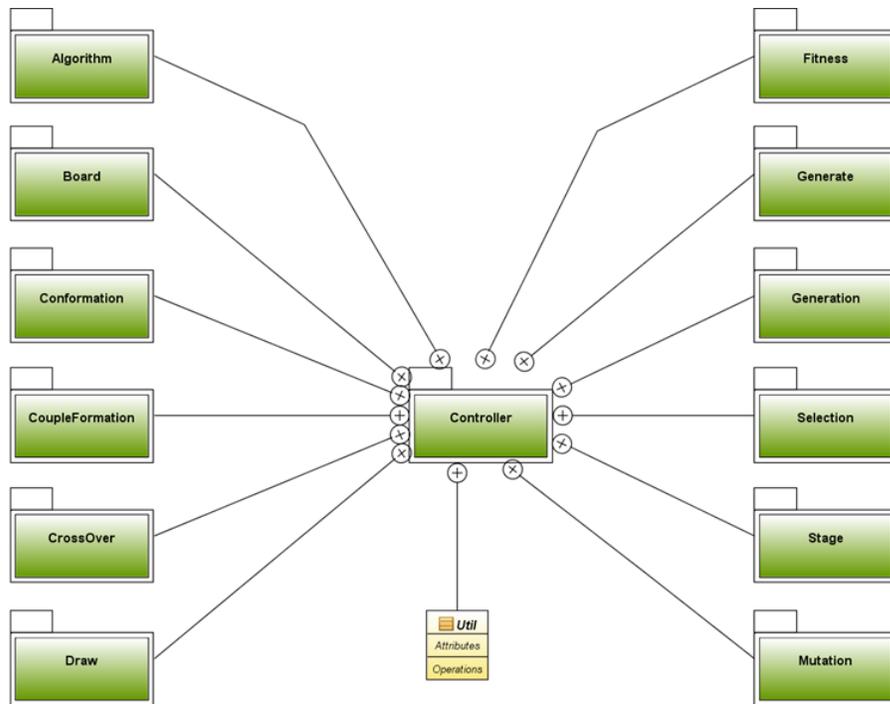


Figura 5.54 Vista Lógica de la plataforma bioinformática Evolution. Detalle C.

Gráficas de Usuario

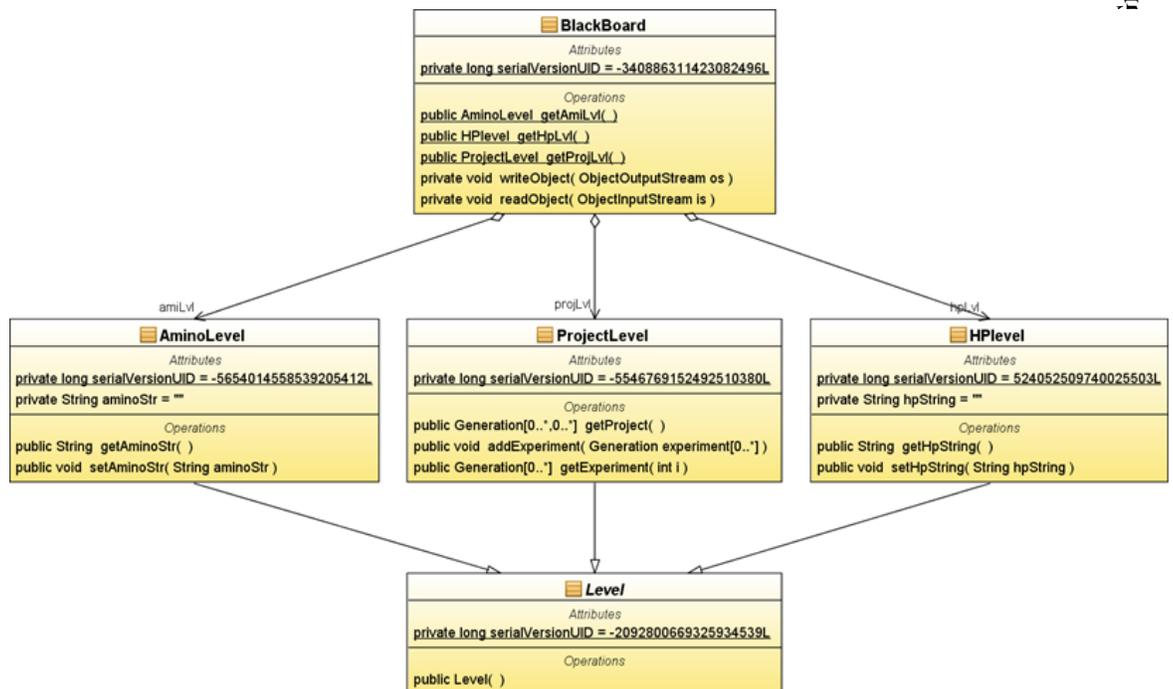
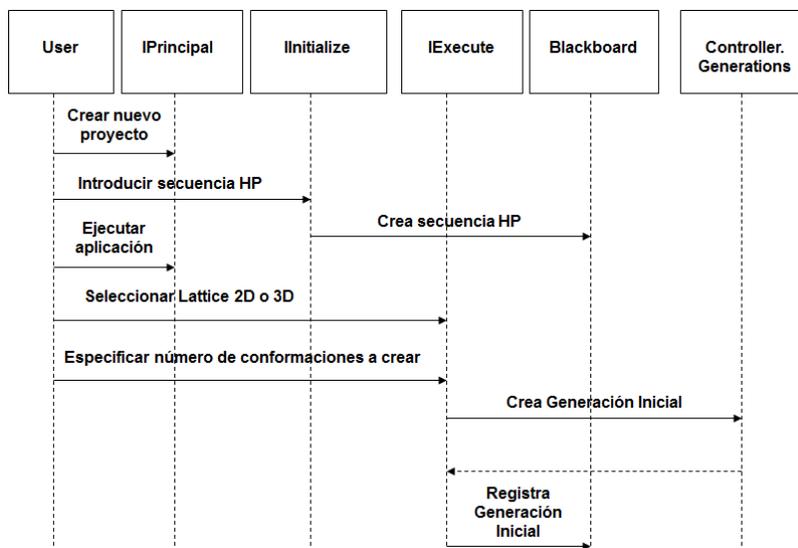
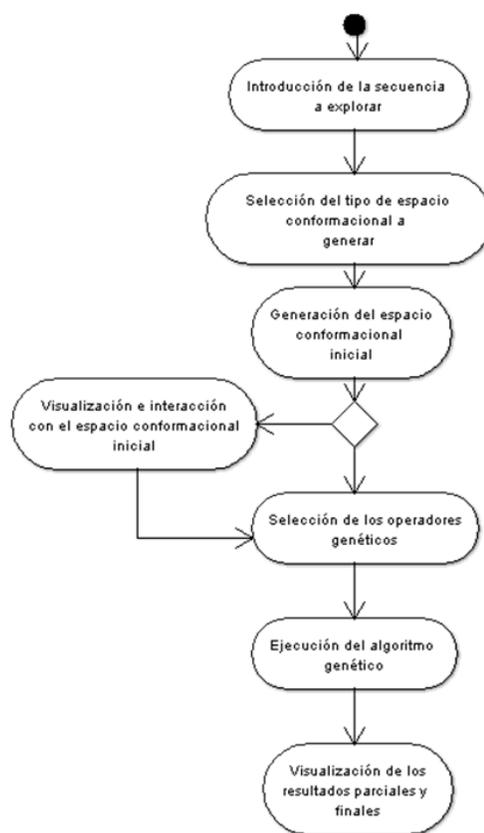


Figura 5.55 Vista Lógica de la plataforma bioinformática Evolution. Detalle D.

Diseño Arqu  
— Casos de



**Figura 5.56** Vista de Proceso de la plataforma bioinformática Evolution. Detalle A.



**Figura 5.57** Vista de Proceso de la plataforma bioinformática Evolution. Detalle B.

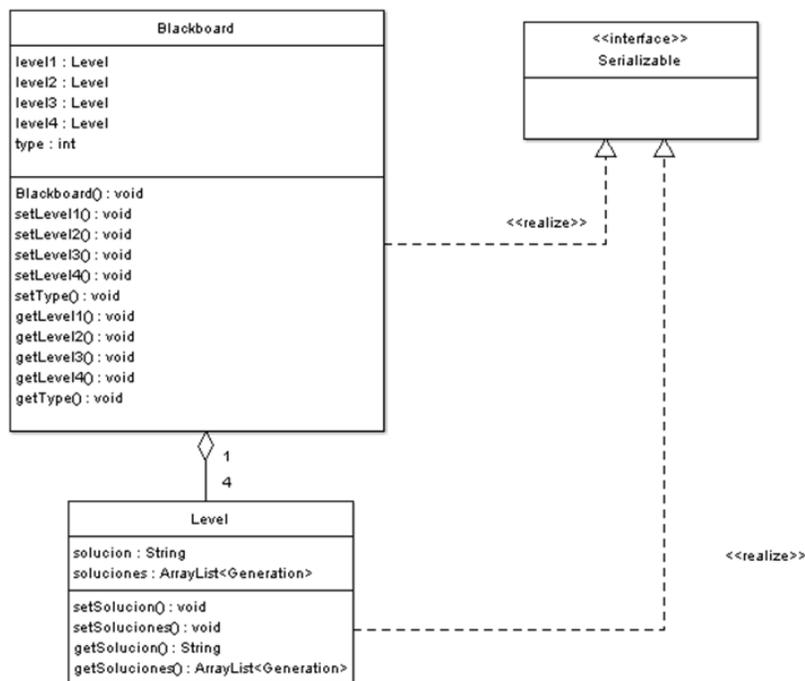


Figura 5.58 Vista de Datos de la plataforma bioinformática Evolution. Detalle A.

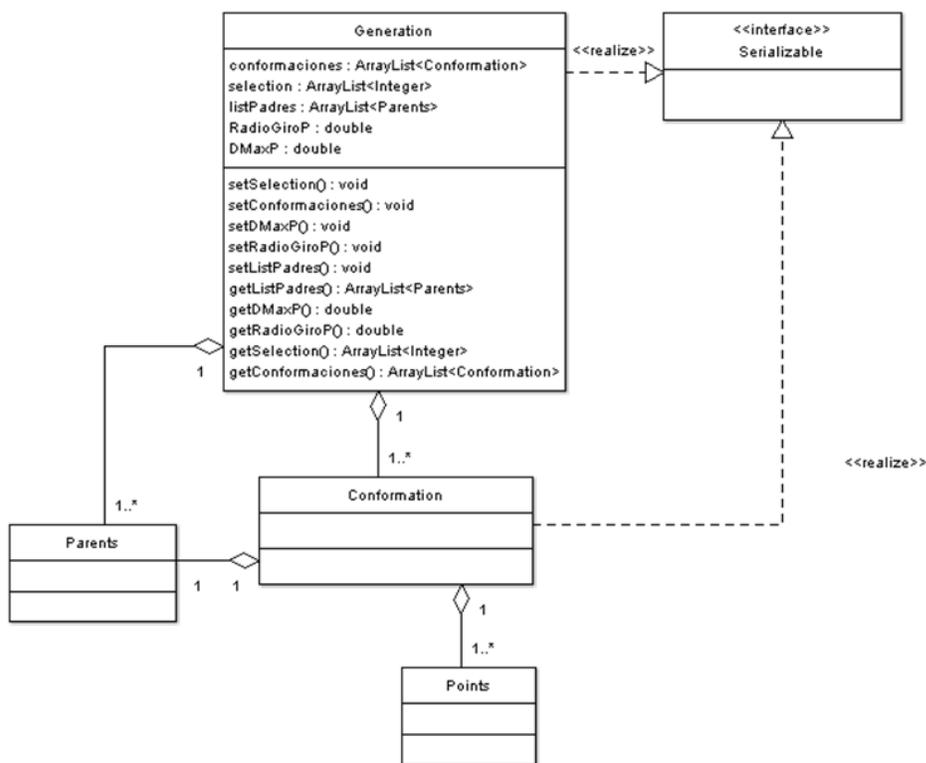
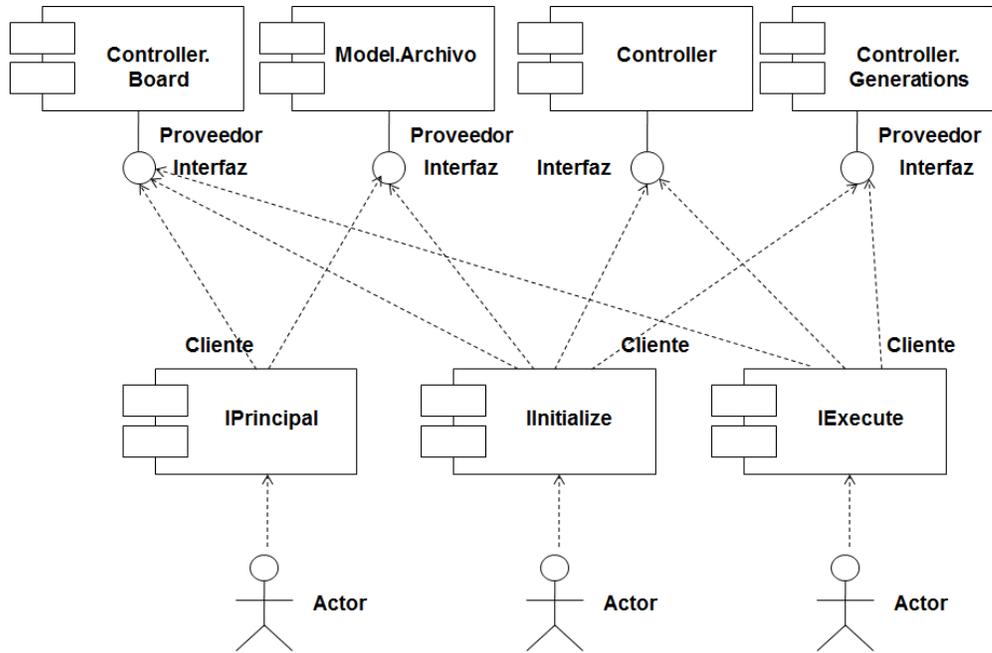
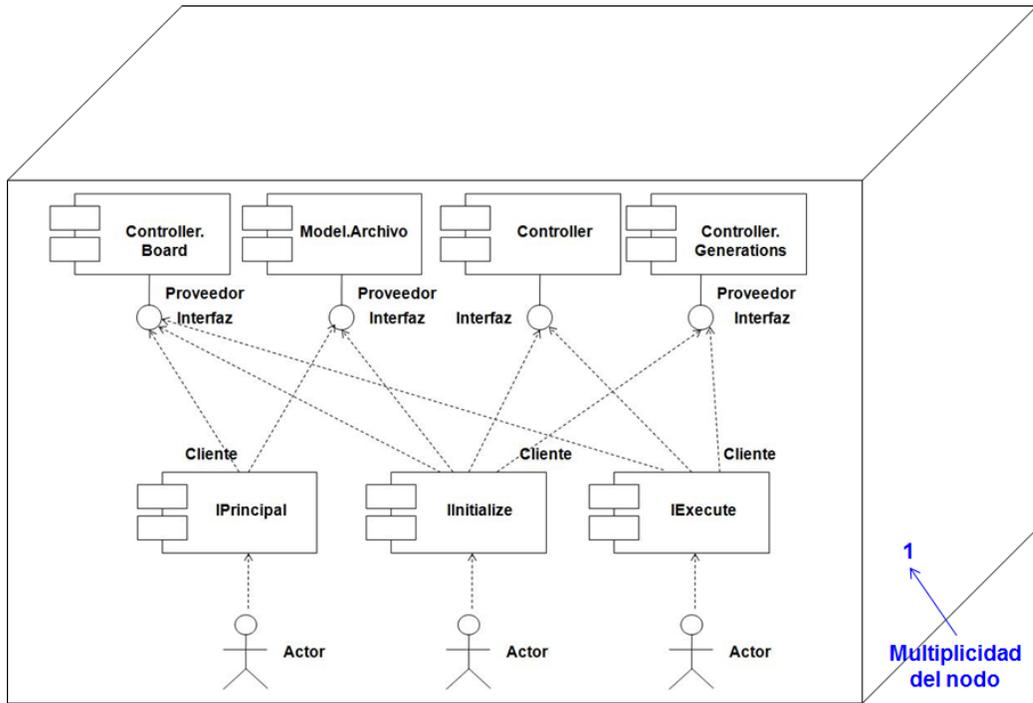


Figura 5.59 Vista de Datos de la plataforma bioinformática Evolution. Detalle B.



**Figura 5.60** Vista de Despliegue de la plataforma bioinformática Evolution. Detalle A.



**Figura 5.61** Vista de Despliegue de la plataforma bioinformática Evolution. Detalle B.

## V.10 Referencias del capítulo

1. Russell, J. (Editor), Cohn, R. (Editor). Model-View-Controller. Bookvika Publishing. 2012.
2. Yah, A. Learning MVC architecture with PHP: to exit beginners, before entering frameworks. Atom Yah. 2017.
3. Pitt, C. Pro PHP MVC. Apress. 2012.
4. González-Pérez, P.P. Notas del Curso Desarrollo de Software a Gran Escala. Material no Publicado. Universidad Autónoma Metropolitana, Unidad Cuajimalpa. 2019.
5. Goldberg, A., Robson, D., Harrison, M.A. (Editor). Smalltalk-80: The Language and its Implementation. Addison-Wesley. 1983.
6. Velthuisen, The Nature and Applicability of the Blackboard Architecture. PTT research. 1992.
7. Jagannathan, V. Blackboard Architectures and Applications. Elsevier. 1989.
8. González-Pérez, P.P. Sistemas Expertos facultativamente asociados en red cooperativa con arquitecturas de pizarrón. Una aplicación en la consulta e interconsulta médica. Tesis de Maestría. Universidad Nacional Autónoma de México. 1995.
9. Soto-Galindo, I., González, P.P. (2010a) Propuesta de un plan para la recuperación de proyectos de software. En González, P.P., Velázquez-Ramírez, I., Franco-Pérez, L. (Eds) *Teorías, Modelos y Aplicaciones de Matemáticas y Computación. Memorias de la 1era y 2da Semana de Computación y Matemáticas Aplicadas SCMA'2008 y SCMA'2009*, Universidad Autónoma Metropolitana, ISBN: 978-607-477-268-5.
10. Soto-Galindo, I., González, P.P. (2010b) Un Plan para la Recuperación de Proyectos de Software inspirado en el Proceso de Diagnóstico Médico. En García Gaona, A.R. y Sánchez Guerrero L. (Eds.) *Desarrollo Tecnológico, Libro Electrónico "XXIII Congreso Nacional y IX Congreso Internacional de Informática y Computación 2010"*, Editorial Alfa Omega, ISBN: 978-607-707-097-9.
11. Sánchez, O. Reingeniería de una plataforma bioinformática para la simulación y experimentación "in silico" de redes de señalización intracelular. Tesis de Maestría. Posgrado en Ciencias Naturales e Ingeniería. Universidad Autónoma Metropolitana, Unidad Cuajimalpa. 2015.
12. González-Pérez, P.P. Notas del Curso Proyecto de Ingeniería del Software I. Material no Publicado. Universidad Autónoma Metropolitana, Unidad Cuajimalpa. 2019.
13. Kruchten, P. The 4+1 View Model of Architecture. IEEE Software 12(6):45-50. 1995.
14. Jacobson, I., Booch, G., Rumbaugh, J. El proceso unificado del desarrollo del software. Pearson Addison-Wesley. 2000.
15. Gómez, M. C., Cervantes, J.: 'User Interface Transition Diagrams for Customer-Developer Communication Improvement in Software Development Projects'. Journal of Systems and Software 2013, 86, (9), pp. 2394-2410.
16. Gómez-Fuentes M. Cervantes-Ojeda J.: 'Application of User Interface Transition Diagrams in the Construction of a Software System'. 7th International

- Conference in Software Engineering Research and Innovation (CONISOFT), IEEE, October 2019, pp. 123-131.
17. Gómez-Fuentes, M. C., Cervantes-Ojeda J.: 'Sequence Diagrams Tailored for Software Design used to Build a Carpooling Management System'.7th International Conference in Software Engineering Research and Innovation (CONISOFT) IEEE,October 2019, pp. 116-122.
  18. González-Pérez, P. P., del Carmen Gómez-Fuentes, M., Velázquez, J. H.: 'A Hybrid Expert System for the Estimation of the Environmental Impact of Urban Development'. Journal of Advances in Mathematics and Computer Science, 2015, pp. 1-17.
  19. <http://www.javaserverfaces.org>.

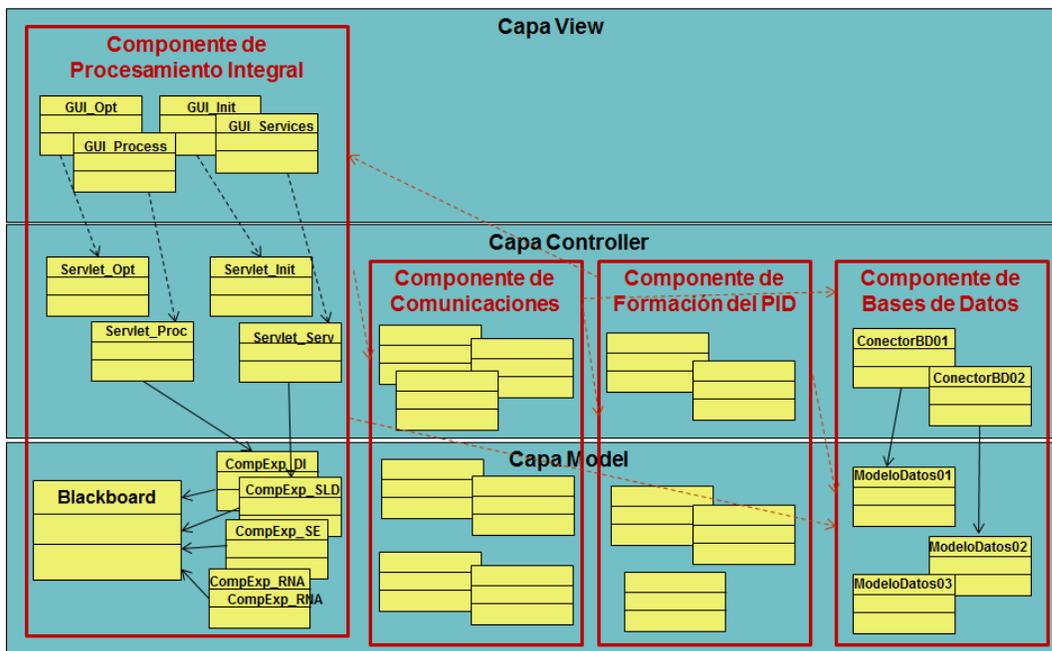
# Capítulo VI Extensiones y Variantes del Modelo Arquitectónico de Sistemas de Software Interactivos

## VI.1 Introducción

En el capítulo anterior nos referimos de forma detallada a la arquitectura de sistemas interactivos Modelo-Vista-Controlador (MVC) [1-3], como una de las soluciones arquitectónicas comúnmente utilizadas en el desarrollo de software a gran escala. En el presente capítulo, revisaremos las principales variantes –y el correspondiente soporte tecnológico– de la arquitectura MVC, tales como:

- Arquitectura Modelo-Vista-Presentador (del inglés, *Model-View-Presenter*).
- Arquitectura Presentación-Abstracción-Control (del inglés, *Presentation-Abstraction-Control* (PAC)).
- Arquitectura Modelo-Vista-Controlador Jerárquico (del inglés, *Hierarchical Model-View-Controller*).
- Arquitectura de cuatro capas Modelo-Vista-Controlador-Servicios.

A diferencia de los anteriores capítulos, en los cuales hemos dedicado el último epígrafe a ejemplificar la aplicación de la temática introducida a través de los casos de estudio SPI y Evolution, en el presente capítulo no dedicaremos un epígrafe final a los casos de estudio, sino que iremos ilustrando como algunas de las variantes de la arquitectura MVC fueron ensayadas durante el diseño arquitectónico del sistema SPI. Para una mejor comprensión de estos enfoques en la arquitectura lógica de SPI, vale la pena partir la arquitectura MVC pura ensayada en SPI e ilustrada en la Figura 6.1.



**Figura 6.1** Arquitectura MVC ensayada durante el diseño arquitectónico del sistema SPI.

## VI.2 Arquitectura Modelo-Vista-Presentador (Model-View-Presenter)

El hecho de que en la literatura se haya reportado hasta la fecha una gran variedad de patrones bajo el nombre *Model-View-Presenter* (MVP) [4, 6] hace difícil resumir tanto la definición como las características de este patrón. Sin embargo, algo debe quedar claro: la inmensa mayoría de (o probablemente todos) los patrones reportados bajo el nombre *Model-View-Presenter* (MVP) constituyen una variante del patrón original *Model-View-Controller* (MVC). Comúnmente, la principal diferencia entre las variantes del MVP y el MVC consiste en que el *Presenter* en el MVP implementa el patrón *Observer* (ver Figura 6.2) entre el *Model* y el *View*. Aquí nos referiremos solo a dos variantes de la arquitectura MVP que, según nuestro punto de vista, resultan entre las más difundidas y utilizadas: el patrón MVP de *Taligent* [4, 6] y el patrón MVP de *Dolphin Smalltalk* [4, 5].

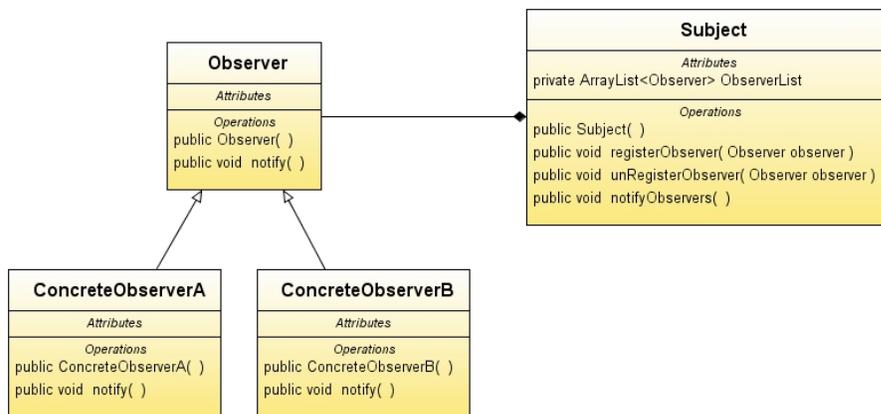


Figura 6.2 Patrón Observer utilizado en la arquitectura MVP.

### VI.2.1 Arquitectura MVP de Taligent

El patrón arquitectónico MVP de Taligent, fue descrito formalmente en la segunda mitad de la década de los 90s por la Corporación Taligent, Inc. [5, 6]. Taligent es una sigla formada de las palabras en inglés Talent e Intelligent. Basado en el modelo de programación homónimo, este patrón arquitectónico extiende las características del patrón original Model-View-Controller de Smalltalk de los años 80's. El patrón Model-View-Presenter de Taligent separa en diferentes partes o componentes las siguientes actividades:

- Aplicación concerniente a los datos.
- Especificación de los datos.
- Manipulación de los datos.
- Coordinación de la aplicación.
- Interacción con el usuario.
- Presentación en las Interfaces gráficas de usuario.

Tomando en cuenta la anterior separación de actividades, el patrón MVP de Taligent define los siguientes componentes:

**View.** Es la representación visual del *Model*. Al igual que en el patrón MVC, el *View* presenta los datos en diferentes formas, pudiendo haber más vistas en la medida en que sea necesario presentar los datos en formas diferentes.

**Interactors.** Son componentes que traducen los eventos originados por el usuario sobre el *View* (por ejemplo, en Java a través de los componentes activos) en operaciones que se ejecutan sobre el *Model*.

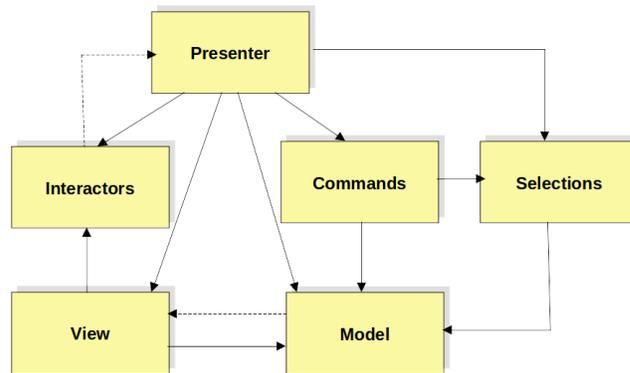
**Model.** Contiene la representación de los datos y las reglas de negocio (funcionalidad del dominio) de la aplicación.

**Commands.** Componentes que definen las operaciones que pueden ser ejecutadas sobre los datos (por ejemplo, búsqueda, recuperación, modificación, almacenamiento, etc.).

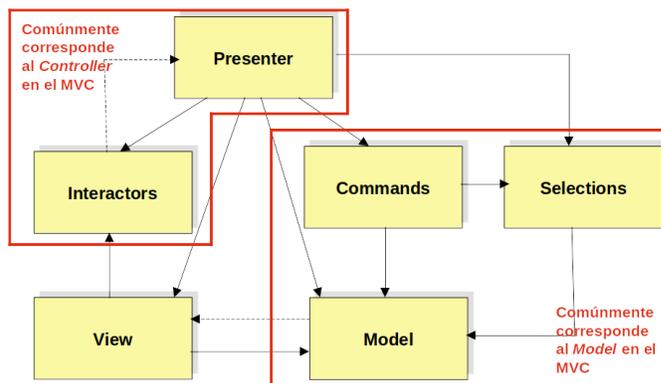
**Selections.** Componentes que seleccionan la parte de los datos dentro del *Model* sobre la cual se operará.

**Presenter.** Componente que controla la interacción global de los restantes componentes de la aplicación.

La Figura 6.3 muestra los componentes e interrelaciones en la arquitectura MVP de Taligent. Por otra parte, en la Figura 6.4 se puede apreciar la correspondencia entre los componentes de la arquitectura MVP de Taligent y la arquitectura MVC.

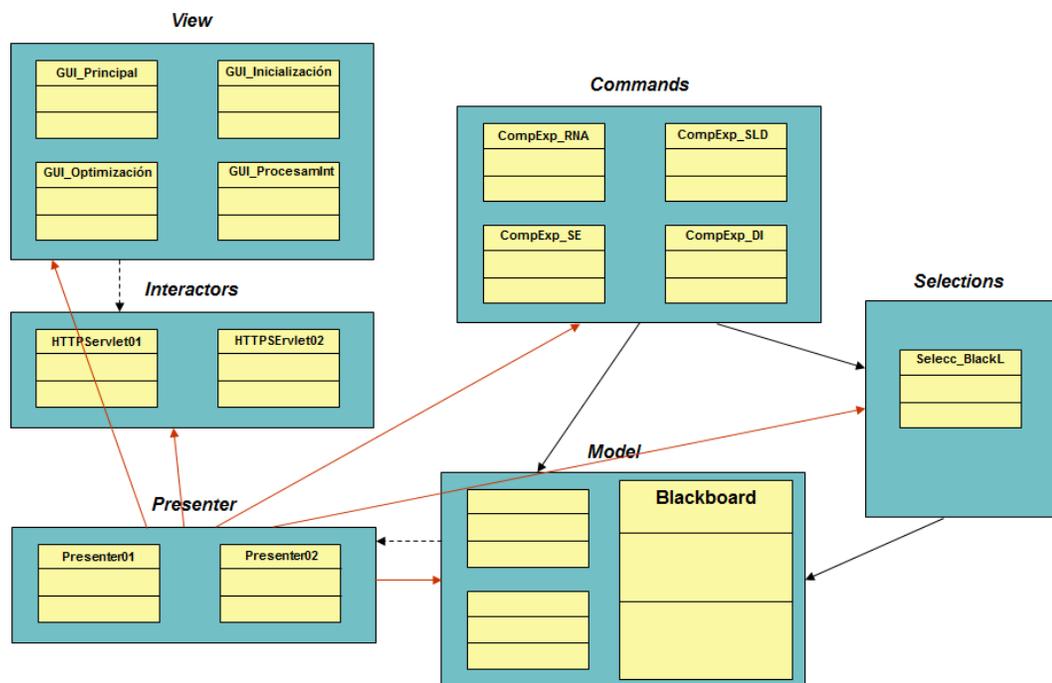


**Figura 6.3** Componentes de la arquitectura MVP de Taligent.



**Figura 6.4** Correspondencia entre los componentes de la arquitectura MVP de Taligent y los componentes de la arquitectura MVC.

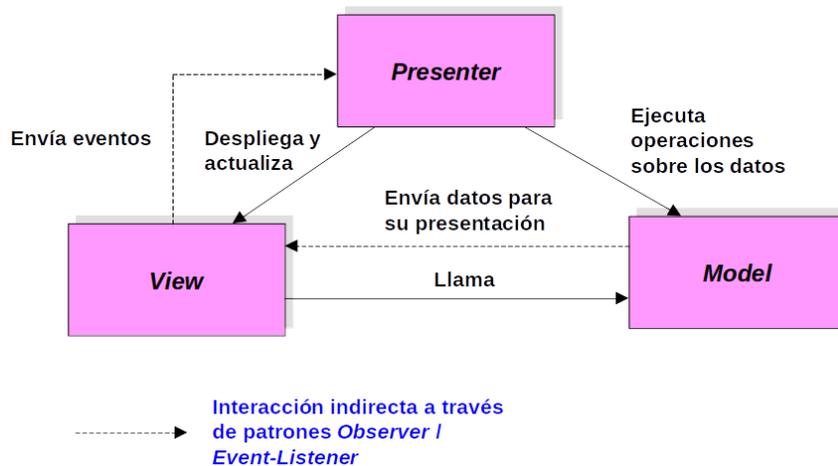
En la Figura 6.5, se ilustra el enfoque MVP de Taligent ensayado durante el diseño arquitectónico del sistema SPI.



**Figura 6.5** Arquitectura MVC ensayada durante el diseño arquitectónico del sistema SPI.

## VI.2.2 Arquitectura MVP de Dolphin Smalltalk

El patrón *Model-View-Presenter* de *Dolphin Smalltalk* [4-5] es una simplificación del patrón *Model-View-Presenter* de *Taligent* y, por lo tanto, una variante muy cercana del patrón original MVC. En el enfoque de *Dolphin Smalltalk* los componentes *Interactors*, *Commands* y *Selections* (característicos del patrón *Model-View-Presenter* de *Taligent*) son omitidos. El papel del componente *Presenter* resulta también simplificado, pasando de un subsistema controlador a un componente que media las actualizaciones en el *Model* requeridas por el *View*. La Figura 6.6 muestra los componentes del patrón arquitectónico MVP de *Dolphin Smalltalk*.



**Figura 6.6** Componentes de la arquitectura MVP de Dolphin Smalltalk.

La responsabilidad de cada una de las tres partes componentes del patrón Model-View-Presenter de Dolphin Smalltalk se resume a continuación:

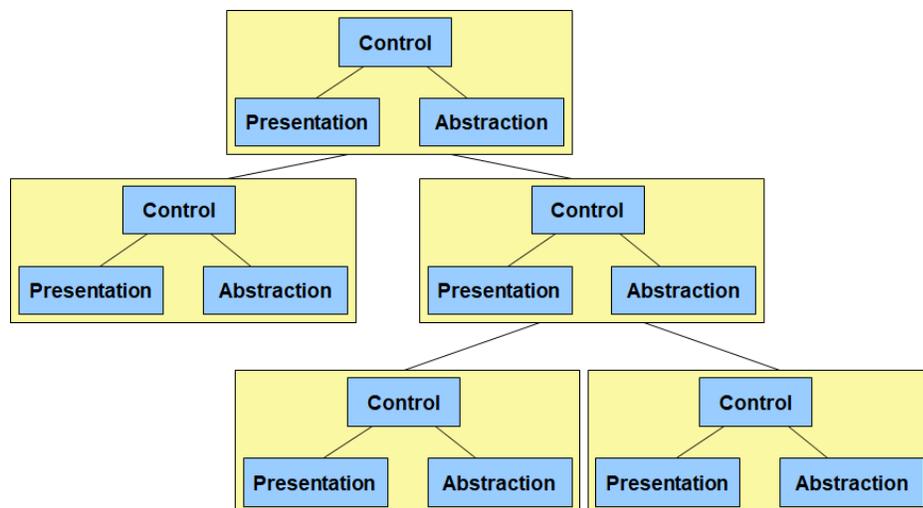
**Model.** Hace referencia a los datos y funcionalidad del dominio de la aplicación.

**View.** Es la representación visual del *Model*. Está compuesto por las interfaces gráficas de usuario y componentes activos requeridos por la aplicación.

**Presenter.** Media la comunicación entre el *View* y el *Model*. Contiene la lógica de la presentación.

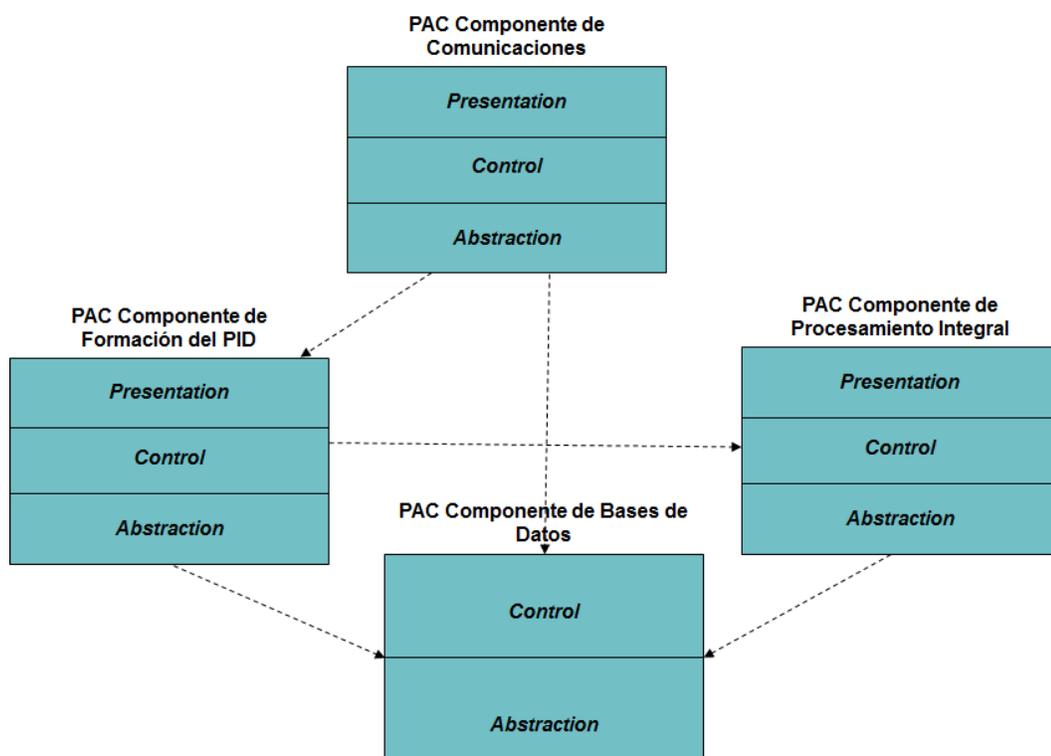
### VI.3 Arquitectura Presentación-Abstracción-Control (Presentation-Abstraction-Control)

El patrón *Presentation-Abstraction-Control* (PAC) es otra variante del patrón *Model-View-Controller* (MVC). La estructura del patrón PAC corresponde a una jerarquía de componentes (también conocidos como agentes) PAC, cada uno de los cuales viene integrado por la tupla: *Presentation-Abstraction-Control*. La comunicación entre los componentes PAC se efectúa solo a través de la parte *Control* de cada tupla. La parte *Presentation* en PAC corresponde a la parte *View* en MVC, mientras que la parte *Abstraction* en PAC corresponde a la parte *Model* en MVC. Otra diferencia del patrón PAC con el patrón MVC consiste en que el patrón PAC no proporciona una comunicación directa entre las partes *Presentation* (View en MVC) y *Abstraction* (Model en MVC). En la Figura 6.7 se pueden apreciar los componentes y relaciones en la arquitectura PAC.

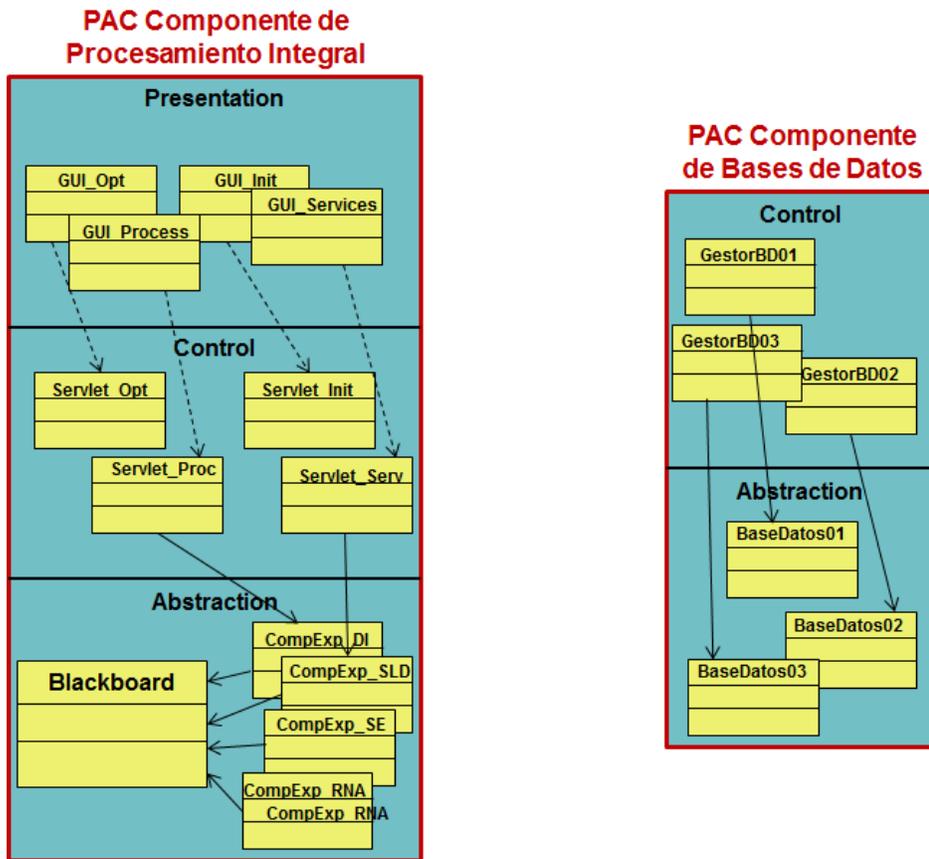


**Figura 6.7** Componentes de la arquitectura PAC.

En las Figuras 6.8 y 6.9, se ilustra el enfoque PAC ensayado durante el diseño arquitectónico del sistema SPI. Este enfoque arquitectónico resultó de gran utilidad, dada la naturaleza orientada a componentes semiindependientes que caracteriza la estructura del sistema SPI.



**Figura 6.8** Arquitectura MVC ensayada durante el diseño arquitectónico del sistema SPI.



**Figura 6.9** Arquitectura MVC ensayada durante el diseño arquitectónico del sistema SPI. Detalle.

## VI.4 Arquitectura Modelo-Vista-Controlador Jerárquico (Hierarchical Model-View-Controller)

El patrón Hierarchical Model-View-Control (HMVC) es una variante del patrón Presentation-Abstraction-Control (PAC), en la cual las partes Presentation y Abstraction pueden comunicarse directamente entre sí, tal y como ocurre en el patrón Model-View-Controller (MVC). La Figura 6.10 ilustra los componentes y relaciones que caracterizan la arquitectura HMVC. Es decir, en HMVC cada componente PAC es modelado como siguiendo el enfoque MVC.

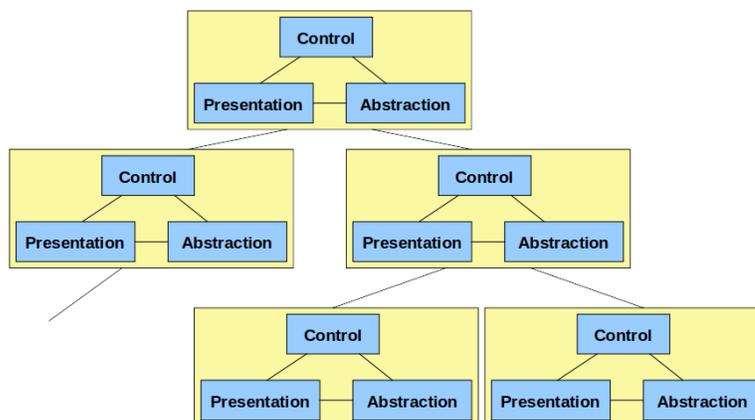


Figura 6.10 Componentes de la arquitectura HMVC.

## VI.5 Arquitectura de cuatro capas Modelo-Vista-Controlador-Servicios

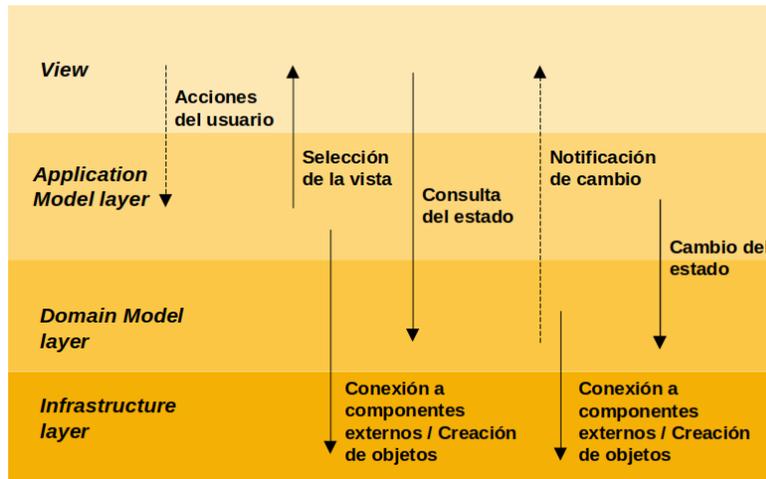
Otra variante de la arquitectura MVC es la conocida como Arquitectura de Cuatro Capas [1-3], la cual se caracteriza por los siguientes aspectos:

- Hereda, extiende y refina alguna de las versiones más ampliamente difundidas de los patrones arquitectónicos Model-View-Controller (MVC) o Model-View-Presenter (MVP).
- Incorpora una nueva capa (la 4ta capa) que generalmente contiene los objetos que representan interfaces o conexiones a entidades externas requeridas por la aplicación, u otros tipos de servicios que no son propiamente los que caracterizan a la funcionalidad o lógica de la aplicación.

Como se puede apreciar en la Figura 6.11, comúnmente, las capas que integran la Arquitectura de Cuatro Capas son Vista, Modelo de la Aplicación, Modelo del Dominio, e Infraestructura, aunque los nombres con los que vienen referidas podrían variar.

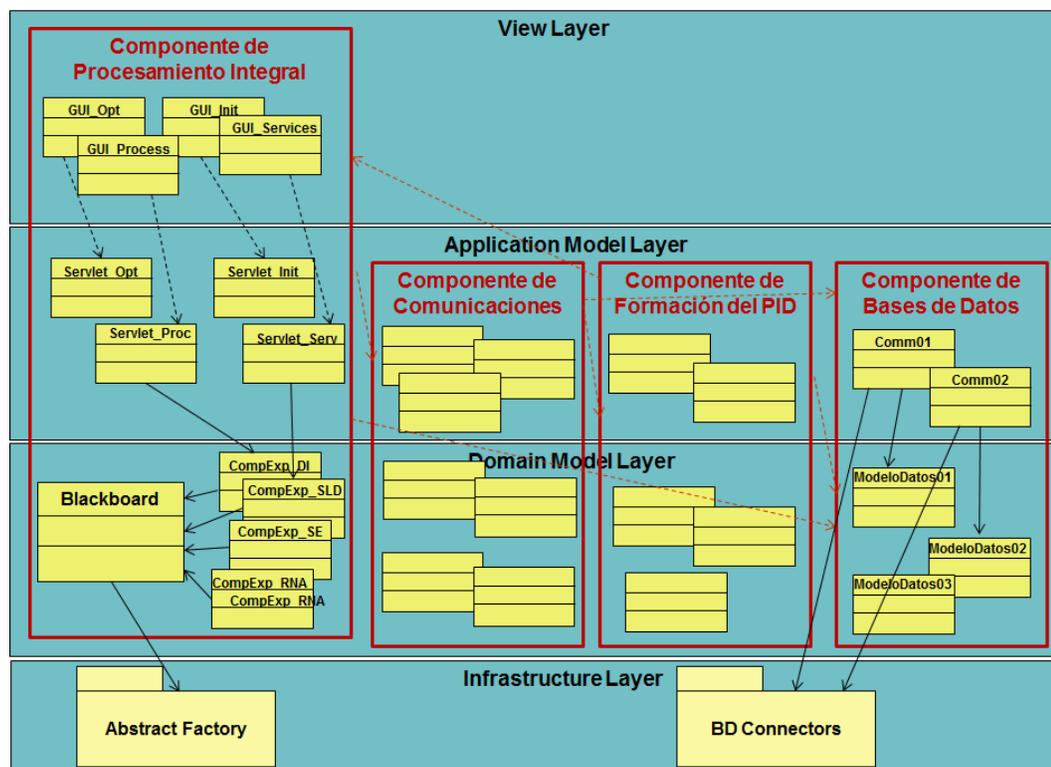
- **Vista (*View layer*)**. Compuesto por las GUI y componentes activos requeridos por la aplicación. Es responsable de la interacción con el usuario.
- **Modelo de la aplicación (*Application Model layer*)**. Media la comunicación entre el *View* y el *Domain Model*. Interactúa con el estrato *Infrastructure*. Contiene la lógica de la aplicación.
- **Modelo del dominio (*Domain Model layer*)**. Contiene los objetos que corresponden a los datos y funcionalidad del dominio de la aplicación.

- **Infraestructura (*Infrastructure layer*)**. Contiene los objetos que representan interfaces o conexiones a entidades externas requeridas por la aplicación (por ejemplo, subsistemas de otras aplicaciones, bases de datos externas, etc.). Como veremos más adelante en algunos de los ejemplos que ilustran este patrón arquitectónico, esta capa resulta de gran utilidad para hospedar patrones de creación, tales como *Abstract Factory*, *Factory Method*, etc.



**Figura 6.11** Las capas y relaciones en la arquitectura de cuatro capas.

En la Figura 6.12, se ilustra el enfoque de la arquitectura de Modelo-Vista-Controlador-Servicios, ensayado durante el diseño arquitectónico del sistema SPI.



**Figura 6.12** Arquitectura Modelo-Vista-Controlador-Servicios ensayada durante el diseño arquitectónico del sistema SPI.

## VI.6 Referencias del capítulo

1. Russell, J. (Editor), Cohn, R. (Editor). Model-View-Controller. Bookvika Publishing. 2012.
2. Yah, A. Learning MVC architecture with PHP: to exit beginners, before entering frameworks. Atom Yah. 2017.
3. Pitt, C. Pro PHP MVC. Apress. 2012.
4. González-Pérez, P.P. Notas del Curso Desarrollo de Software a Gran Escala. Material no Publicado. Universidad Autónoma Metropolitana, Unidad Cuajimalpa. 2019.
5. Goldberg, A., Robson, D., Harrison, M.A. (Editor). Smalltalk-80: The Language and its Implementation. Addison-Wesley. 1983.
6. Potel, M. MVP: Model-View-Presenter. The Taligent Programming Model for C++ and Java. VP & CTO Taligent, Inc. 1996.

# Capítulo VII Modularización y Diseño de Componentes

## VII.1 Modularización, cohesión y acoplamiento

Otro de los aspectos que caracteriza el desarrollo de software a gran escala es la necesidad de garantizar un alto nivel de cohesión dentro de cada módulo y un bajo nivel de acoplamiento entre dichos módulos. La modularización [1-5] se refiere a la descomposición de un sistema en un conjunto de componentes o módulos. El principio del diseño modular consiste en descomponer el sistema en módulos. Cuando usamos el paradigma orientado a objetos, un módulo comúnmente consiste de un conjunto de clases e interfaces estrechamente relacionadas entre sí. Es posible también, mediante el uso del patrón de diseño de “fachada”, hacer que todas las clases de un módulo queden detrás de dicha fachada. Así, el módulo es visto desde afuera como un solo ente evitándose la necesidad de que los demás módulos deban estar enterados de la existencia de cada clase dentro del módulo. A nivel del análisis, el diseño del módulo puede implicar la creación de elementos tales como diagramas de casos de uso, etcétera, específicos para el módulo. Existen dos criterios clave para valorar qué tan apropiada es la propuesta de un diseño modular:

- **Cohesión.** Se refiere al grado de relación entre las tareas dentro de un mismo módulo.
- **Acoplamiento (interdependencia).** Se refiere al grado de dependencia entre los diferentes módulos para poder llevar a cabo sus tareas.

Un buen diseño modular implica una alta cohesión y un bajo acoplamiento. Un módulo es un mecanismo que permite agrupar clases, interfaces, etc. en unidades de nivel superior. Uno de los criterios más comúnmente usados para agrupar las clases (y otros elementos del sistema) en módulos es la cohesión. Un diagrama de módulos debe mostrar los módulos y la interdependencia o acoplamiento entre los mismos mediante líneas que indican la presencia de comunicación entre módulos. Un diagrama de módulos puede ser visto, hasta cierto punto, como un diagrama de clases a un nivel de granularidad grueso.

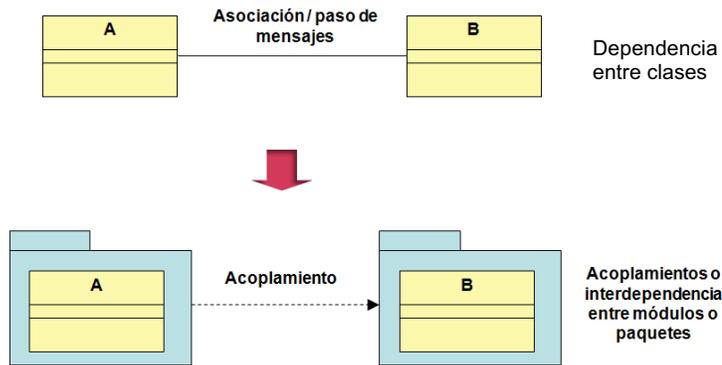
Entre dos clases existe dependencia si:

- Una clase envía un mensaje a otra clase (llama a sus métodos).

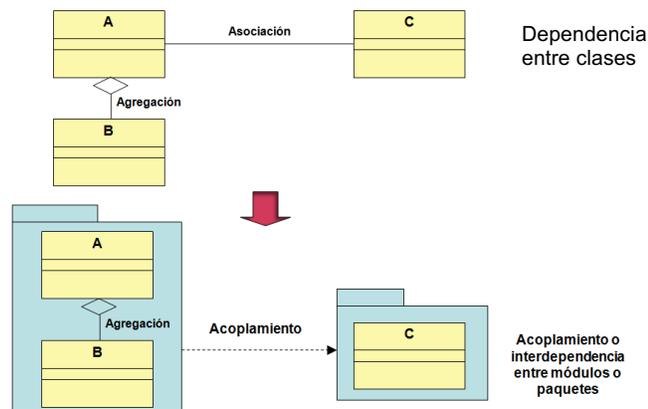
- Existe una relación de agregación/composición entre ambas clases: una clase contiene a la otra como parte de sus atributos.
- Una clase hace referencia a la otra como parámetro para una operación.

Entre el módulo A y el módulo B existe un acoplamiento o interdependencia si existe cualquier tipo de dependencia entre una clase del módulo A y una clase del módulo B.

Las Figuras 7.1 y 7.2 ilustran como la dependencia entre clases se traduce en el acoplamiento entre los módulos que contienen dichas clases.



**Figura 7.1** De la dependencia entre clases al acoplamiento entre módulos. Caso A.

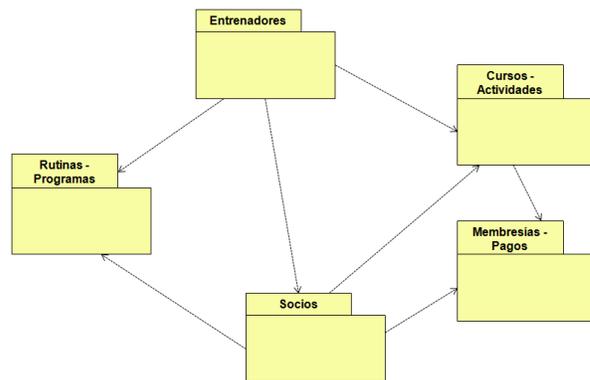


**Figura 7.2** De la dependencia entre clases al acoplamiento entre módulos. Caso B.

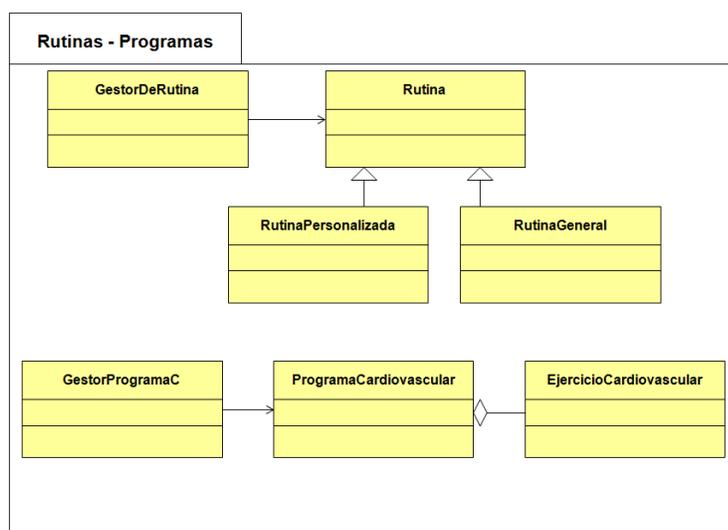
Los módulos permiten el agrupamiento de elementos del diseño (clases, interfaces, etc.) que están fuertemente relacionados entre sí. Un módulo puede contener clases, interfaces y a su vez, otros paquetes. Como ya nos referimos, dos características deseables para los módulos son la alta cohesión interna y el débil acoplamiento externo. Un módulo muestra cohesión cuando los elementos que

agrupa están fuertemente relacionados. Entre dos módulos existe un débil acoplamiento cuando sus dependencias son mínimas.

Las Figuras 7.3 y 7.4 muestran algunos aspectos de la modularización efectuada durante el diseño del sistema Club Deportivo Virtual *Sport Planet*. Mientras que la Figura 7.3 exhibe la modularización inicial propuesta, la Figura 7.4 muestra la cohesión entre las clases que conforman el módulo Rutinas – Programas.



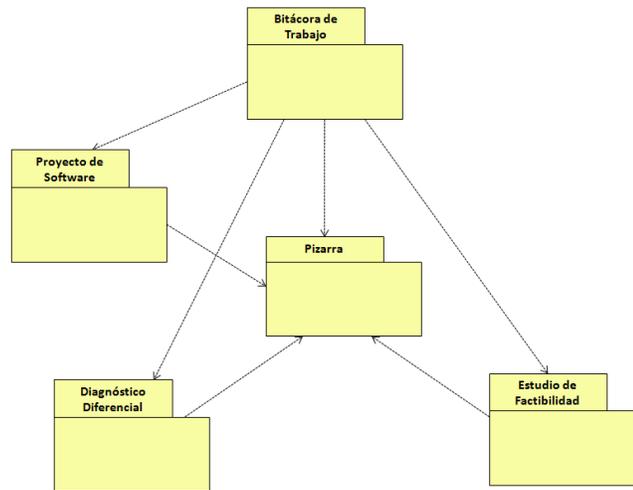
**Figura 7.3** Diagrama de paquetes que representa la modularización inicial propuesta en el sistema Club Deportivo Virtual Sport Planet.



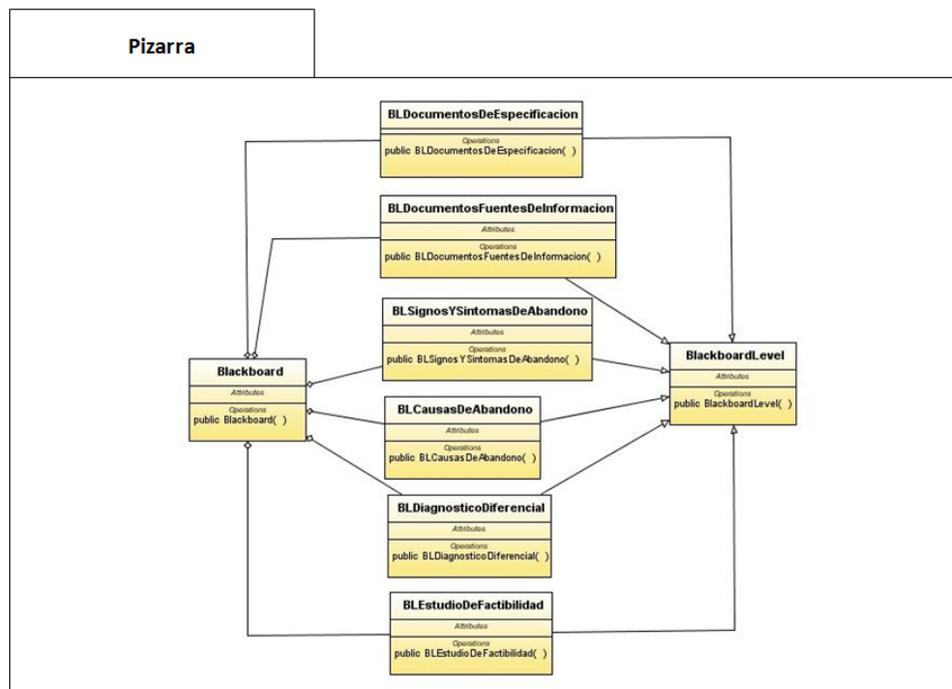
**Figura 7.4** Relaciones internas que exhibe el módulo “Rutinas - Programas” en el sistema Club Deportivo Virtual Sport Planet. Se ha utilizado la notación de diagramas de paquetes.

Otro ejemplo de modularización, como principio clave del desarrollo de software a gran escala, es ilustrado en las Figuras 7.5, 7.6 y 7.7, donde se muestran aspectos de la modularización del Sistema para la Recuperación de Proyectos de Software RPS-GS. La Figura 7.5 muestra la modularización inicial propuesta para la lógica de la aplicación, considerando las funcionalidades o prestaciones a nivel superior que debe satisfacer el sistema RPS-GS. La Figura 7.6 exhibe las relaciones entre las clases contenidas en el módulo “Pizarra”. Finalmente, la Figura 7.7 presenta la capa

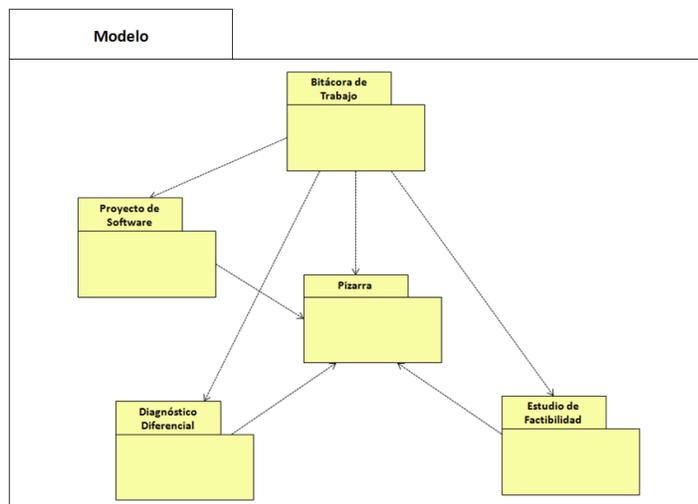
Modelo, de la arquitectura interactiva del sistema RPS-GS, como un módulo contenedor de todos los módulos que intervienen en la lógica del sistema.



**Figura 7.5** Diagrama de paquetes que representa la modularización inicial propuesta en el sistema RPS-GS.



**Figura 7.6** Relaciones internas que exhibe el módulo “Pizarra” en el sistema RPS-GS. Se ha utilizado la notación de diagramas de paquetes.



**Figura 7.7** La capa Modelo como módulo contenedor de otros módulos en el sistema RPS-GS. Se ha utilizado la notación de diagramas de paquetes.

## VII.2 Diseño de componentes

Como resultado de la modularización se genera un grupo de componentes (módulos) y relaciones (interdependencia o cohesión) entre dichos componentes. Cada uno de estos componentes requerirá de un ulterior trabajo de diseño detallado, que abarque desde diferentes vistas (lógica, de proceso, de desarrollo, etc.) la interfaz del componente, los elementos que contiene el componente (clases, interfaces de implementación, etc.), las relaciones entre dichos elementos, así como el nivel de detalle (atributos y métodos) de cada uno de los elementos que integran el módulo.

El diseño de componentes (modelo de diseño o diseño detallado) parte del diseño arquitectónico, y modela en detalle cada una de las partes o componentes (módulos, clases, funciones, procedimientos, etc.) que integran la arquitectura propuesta, aplicando un diseño modular efectivo. En el diseño de componentes se diseña de forma detallada, a través de los modelos y diagramas más apropiados, la comunicación y dependencia entre los módulos diseñados.

Aunado a lo anterior, el diseño de componentes debe verificar que cada uno de los componentes diseñados a detalle satisface los requerimientos funcionales especificados en la etapa de análisis y, por lo tanto, las necesidades y expectativas del cliente. El diseño de componentes debe proporcionar una especificación de diseño a un nivel de detalle tal, que el proyecto quede listo para la fase de implementación/codificación.

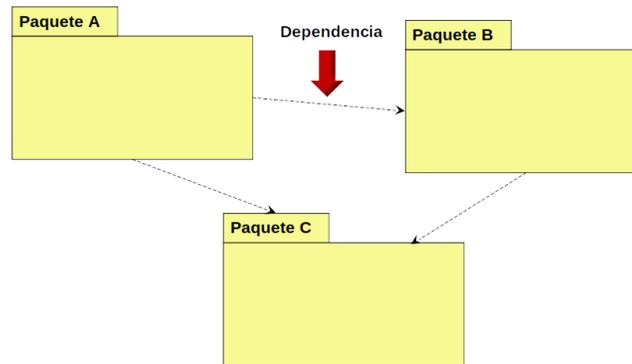
El modelo de diseño se refiere al conjunto de diagramas que describen el diseño lógico. Es la solución lógica basada en el paradigma orientado a objetos. El modelo de diseño incluye, entre otros, los siguientes diagramas:

- Diagramas de paquetes.
- Diagramas de clases.
- Diagramas de interacción.
- Diagramas de estados.
- Diagramas de actividades

### VII.2.1 Diagramas de paquetes

En el Epígrafe 7.1 de este capítulo, ya utilizamos los diagramas de paquetes para reflejar decisiones de modularización en el diseño, aunque no nos habíamos referido a éstos tal como hemos hecho con los diagramas de clases, de interacción (secuencia y colaboración), de transición de estados y de actividades.

Un diagrama de paquetes exhibe las dependencias entre los paquetes de clases que componen el modelo de diseño. El paquete “A” establece una relación de dependencia con el paquete “B”, si al menos un elemento del paquete “A” requiere de al menos un elemento del paquete “B”. Estos elementos son comúnmente clases. Los elementos de un diagrama de paquetes son principalmente los paquetes y las relaciones de dependencia entre los mismos. Nótese que un paquete comúnmente representa un módulo o componente del sistema. La figura 7.8 ilustra una representación genérica de un diagrama de paquetes.



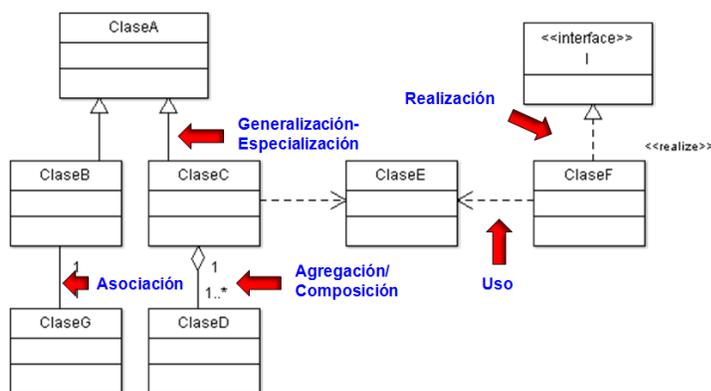
**Figura 7.8** Representación genérica de un diagrama de paquetes.

En las Figuras 7.3 a la 7.7 ya ilustramos ejemplos del uso de los diagramas de paquetes, que reflejan decisiones de modularización tomadas durante el diseño arquitectónico y detallado de sistemas de software.

### VII.2.2 Diagramas de clases

Un diagrama de clases [6,7] modela los conceptos del dominio de la aplicación (clases y relaciones entre clases), así como los nuevos conceptos que surgen durante el diseño detallado (interfaces, clases de implementación, gestores de datos, por ejemplo.). Un diagrama de clases describe los tipos de objetos (clases) que

componen el sistema y los diferentes tipos de relaciones estáticas que existen entre éstos. Como se ilustra en la Figura 7.9, los componentes principales del diagrama de clases son las clases y todas las posibles relaciones existentes entre éstas (asociación, generalización-especialización, agregación/composición, realización y uso).



**Figura 7.9** Diagrama de clases genérico que ilustra algunas de las principales relaciones que se establecen entre las clases.

Además de los tipos de relaciones existentes, un diagrama de clases muestra los atributos y operaciones de una clase, así como las restricciones impuestas en dependencia del tipo de relación.

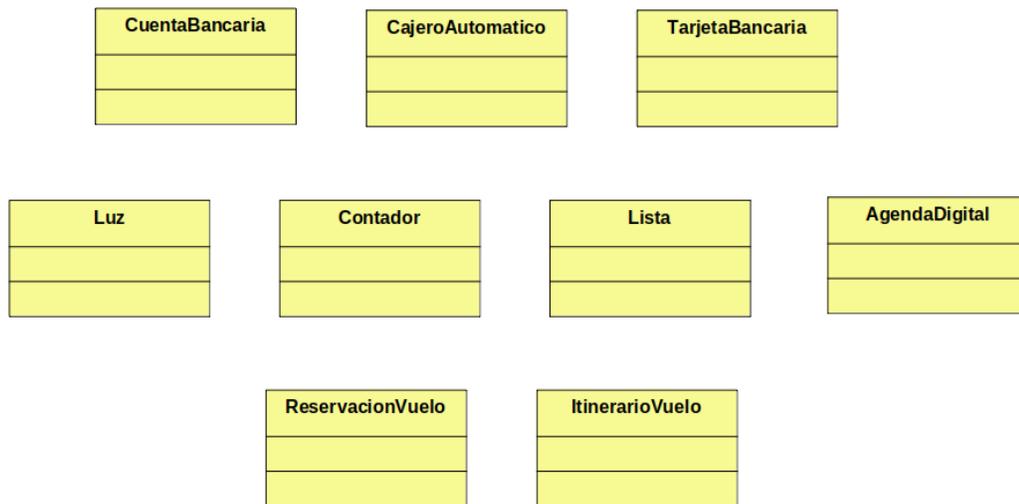
Una clase define la estructura y el comportamiento de los objetos que son instancias de esa clase. En los lenguajes orientados a objetos, cada objeto es una instancia (construcción concreta) de una clase. La clase es la descripción de la estructura y del comportamiento que caracteriza a todos los objetos instancias de esa clase. Una clase es el *template* (molde) utilizable para la construcción de los objetos. Especificamos la estructura y el comportamiento de un objeto describiendo la clase de la cual el objeto es instancia. Dada una clase, podemos *instanciar*, crear dinámicamente todos los objetos que deseemos (limitado solo por la memoria disponible), como entidades separadas e independientes.

Las clases no existen como entidades concretas, son solo puras descripciones. Aunque en realidad en algunos lenguajes como *Smalltalk* y Java las mismas clases pueden ser representadas como objetos. A diferencia de las clases, los objetos “existen”, creados con estructura y comportamiento definidos por la clase a la que pertenecen, y diversificados por el estado que cambia por las interacciones que tienen en el curso del tiempo. La Figura 7.10 ilustra varios ejemplos de clases identificadas en varios dominios de aplicación, entre los que se encuentran un sistema Web bancario y un sistema Web de reservaciones y ventas de una aerolínea.

Una clase viene caracterizada por un nombre y por la descripción de sus elementos, es decir:

- **Atributos** (*definidos también como data member*). Definen las estructuras de datos internas al objeto, por lo tanto, el estado del objeto.
- **Métodos** (definidos también como funciones miembros). Definen el comportamiento del objeto, las operaciones que éste puede ejecutar.

Entre los métodos, existen algunos especiales –definidos como constructores– que vienen invocados solo en el momento de la creación dinámica de un objeto, para inicializar dicho objeto oportunamente. Cada objeto creado dinámicamente como instancia de la clase C tendrá la estructura y el comportamiento descrito por la clase C. Su estado dinámico después variará en dependencia de las interacciones que tendrá en el transcurso de su existencia.



**Figura 7.10** Ejemplos de clases identificadas en varios dominios de aplicación.

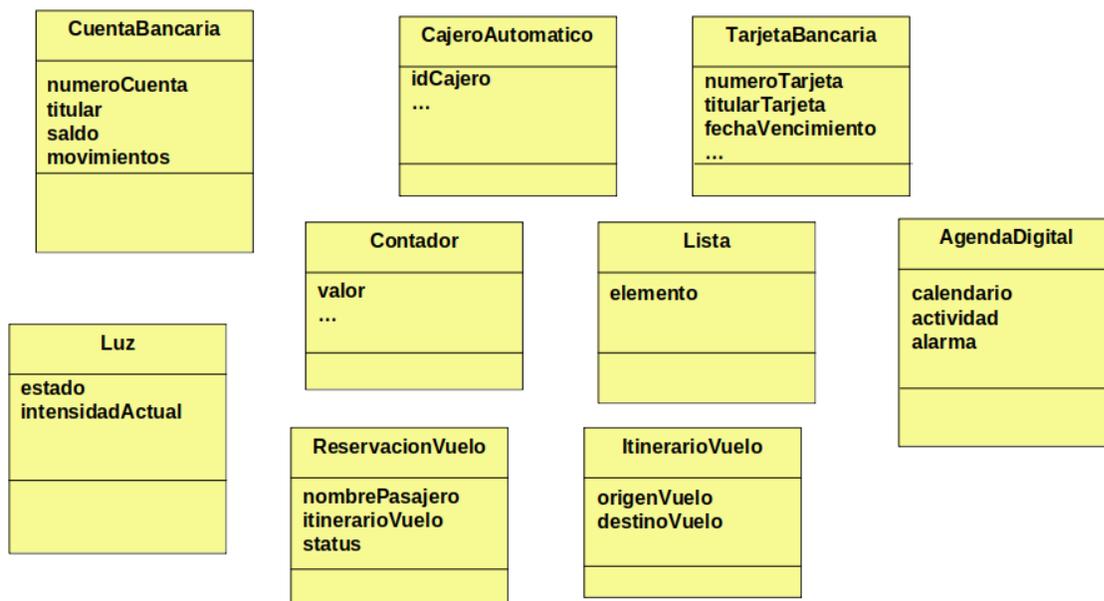
Los atributos o campos (también conocidos como *data member*) constituyen la estructura de datos propia de la clase, estructura que comúnmente deseamos mantener privada, no accesible directamente desde el exterior (principio de *information hiding*). En Java, la definición de campos consiste en una lista de definiciones del tipo:

```
private <TipoCampo> <NombreCampo>;
```

De este modo, definimos un campo (privado al objeto, no visible desde el exterior), con el nombre y tipo especificados. Por convención, los nombres de los campos deben ser escritos en minúsculas. Para nombres compuestos, es posible iniciar las palabras sucesivas con mayúscula.

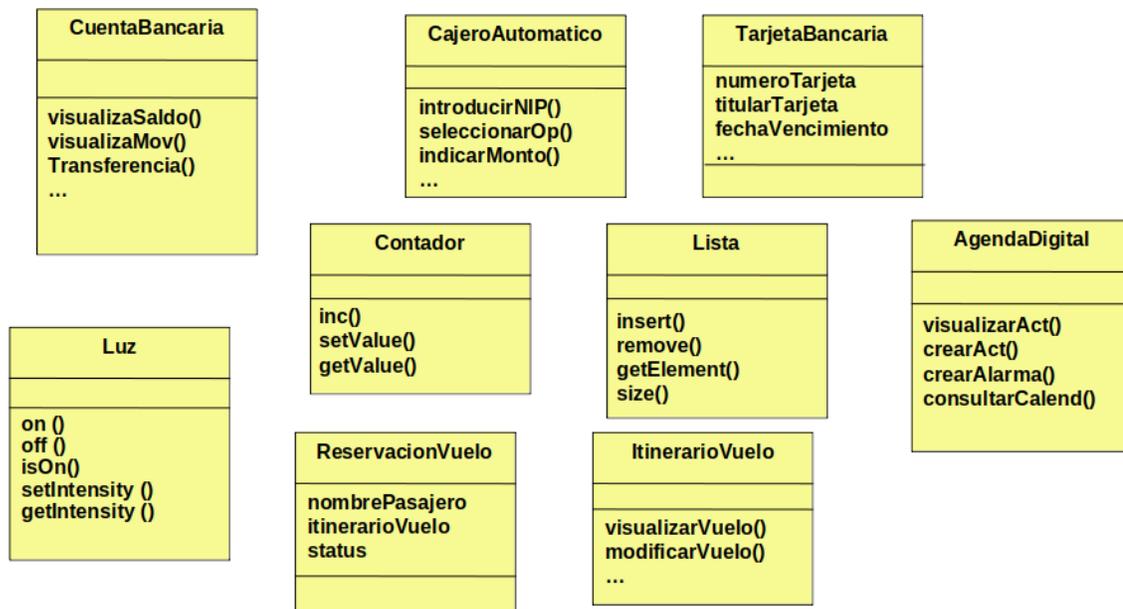
Para cada objeto, instancia de una clase, los atributos constituyen el estado del objeto. El estado típicamente evoluciona en la misma medida en que el objeto interactúa con el mundo externo, es decir, cuando vienen invocados métodos/operaciones del objeto que cambian tal estado. El estado puede estar a su vez constituido por un conjunto de objetos, es decir, un campo puede ser a su vez

un objeto. El estado de un objeto permanece oculto al usuario del objeto. Los atributos – privados – no son jamás accedidos directamente por quien interactúa con el objeto, a quien queda completamente oculto el modo en el cual tal estado ha sido implementado (cuales atributos, que tipos, etc.). La Figura 7.11 muestra la identificación de algunos atributos para las clases relacionadas en la Figura 7.10.



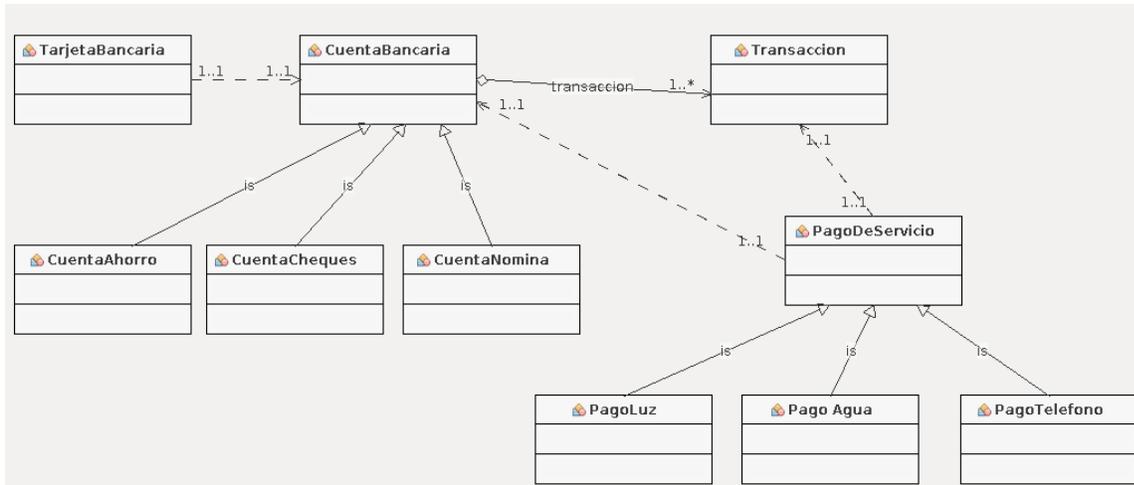
**Figura 7.11** Identificación de atributos para las clases ilustradas en la Fig. 7.10.

Los métodos constituyen el comportamiento de los objetos instancias de la clase. Es decir, las operaciones que el objeto instancia de la clase puede ejecutar. Entre los métodos, existen algunos especiales – definidos constructores – que vienen invocados solo en el momento de la creación dinámica de un objeto, para inicializarlo oportunamente. Cada objeto creado dinámicamente como instancia de la clase C tendrá la estructura y el comportamiento descrito por la clase C. Su estado dinámico después variará en dependencia de las interacciones que tendrá en el transcurso de su existencia. Es decir, a través de las invocaciones a aquellos métodos que pueden cambiar su estado (sus atributos). En la Figura 7.12 se muestra la identificación de métodos para algunas de las clases ilustradas en la Figura 7.10.

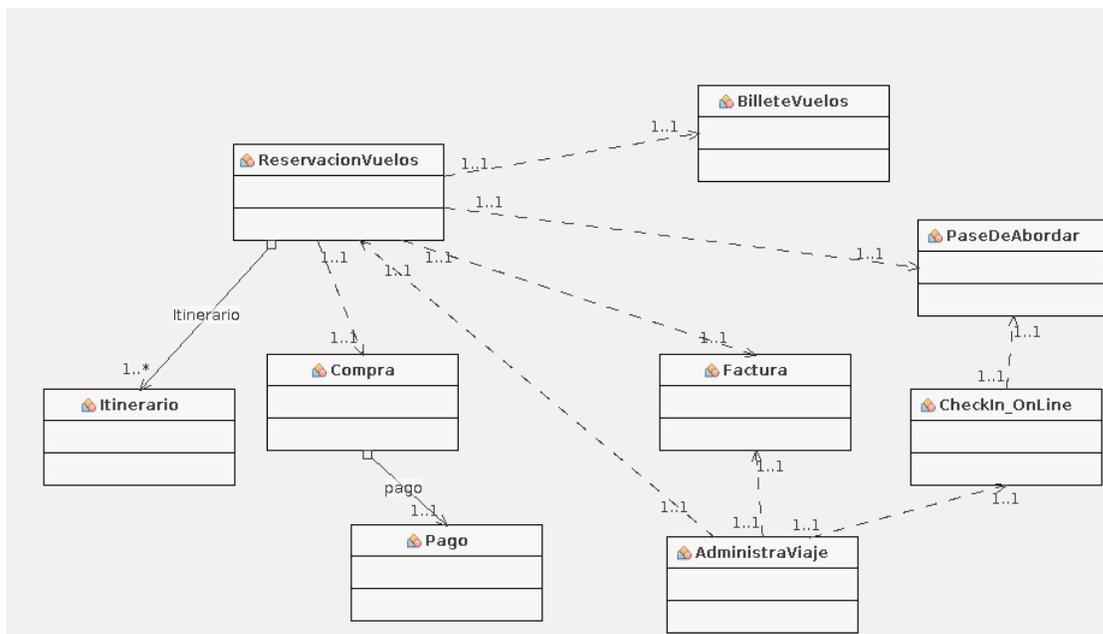


**Figura 7.12** Identificación de métodos para algunas de las clases ilustradas en la Figura 7.10.

Las Figuras 7.13 y 7.14 muestran el uso de los diagramas de clase, durante el diseño de dos sistemas de software. Como se puede apreciar en ambas figuras, los diagramas de clases involucran algunas de las clases ya identificadas en las Figuras 7.10 a la 7.12. Por una parte, la Figura 7.13 muestra una versión inicial del diagrama de clases del sistema Web Portal Bancario. Nótese como en dicho diagrama se incluyen las clases “*CuentaBancaria*” y “*TarjetaBancaria*”, previamente identificadas en las figuras 7.10 a la 7.12. Por otra parte, la Figura 7.14 proporciona una versión de un diagrama de clases del sistema Web de reservaciones y ventas de una aerolínea. Nótese que dicho diagrama incluye las clases “*ReservacionVuelo*” e “*ItinerarioVuelo*”, previamente identificadas en las Figuras 7.10 a la 7.12.

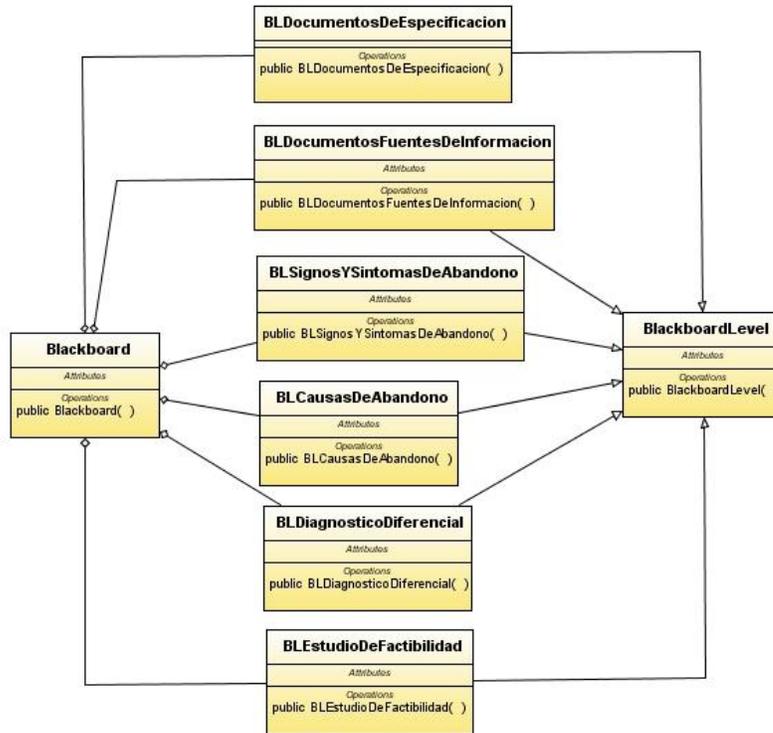


**Figura 7.13** Diagrama de clases que ilustra las principales clases del dominio y relaciones del sistema Web bancario.

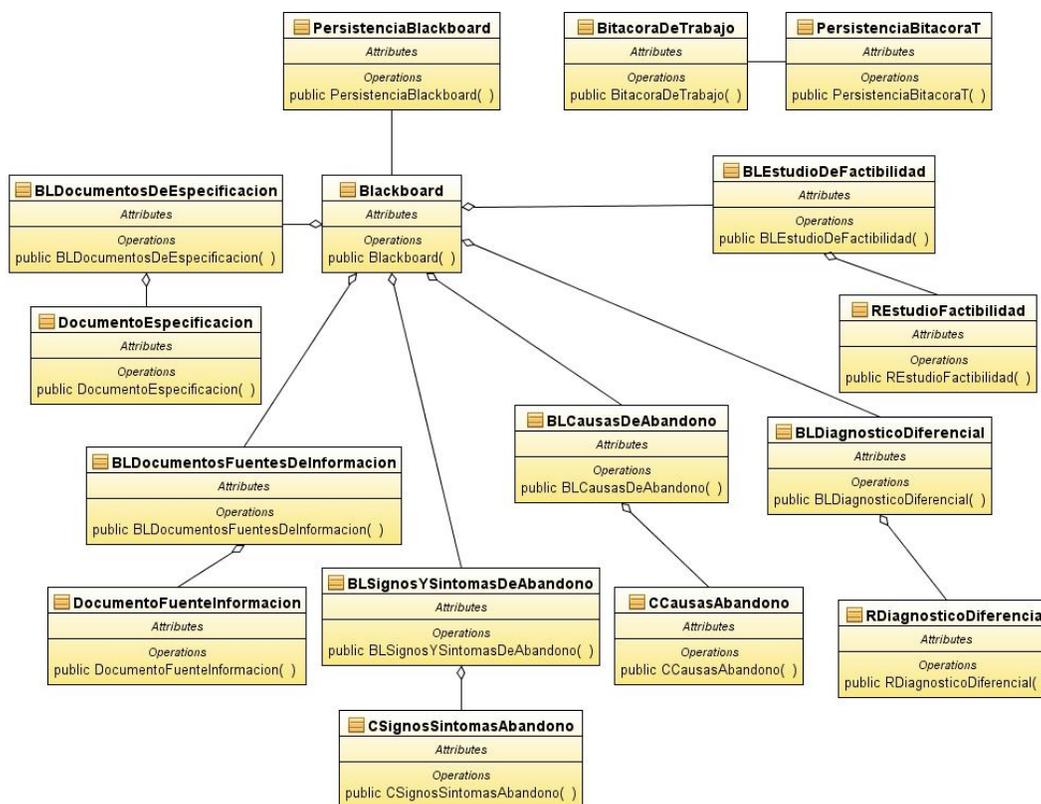


**Figura 7.14** Diagrama de clases que ilustra las principales clases del dominio y relaciones del sistema Web de reservaciones y ventas de una aerolínea.

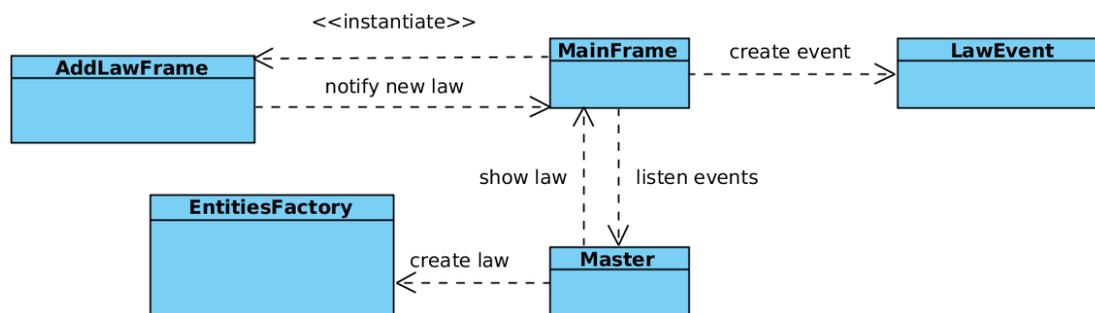
Las Figuras 7.15 a la 7.19 ilustran otros ejemplos de diagramas de clases más elaborados, correspondientes al diseño de sistemas de software. Las Figuras 7.15 y 7.16 ilustran parte del modelado de las principales clases de la lógica del sistema RPS-GS. Por otra parte, las Figuras 7.17 a la 7.19 exhiben parte de la vista estática del diseño de la plataforma bioinformática Cellulat.



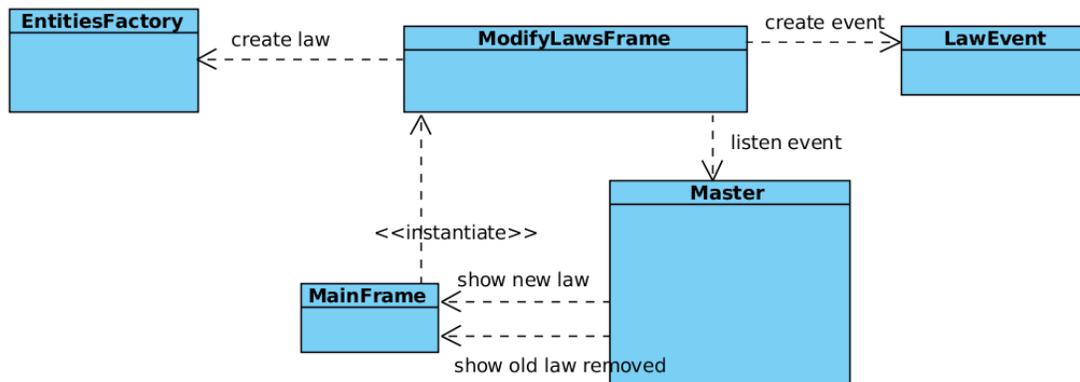
**Figura 7.15** Diagrama de clases que ilustra las principales clases de la lógica del Sistema de Recuperación de Proyectos de Software RPS-GS. Vista A.



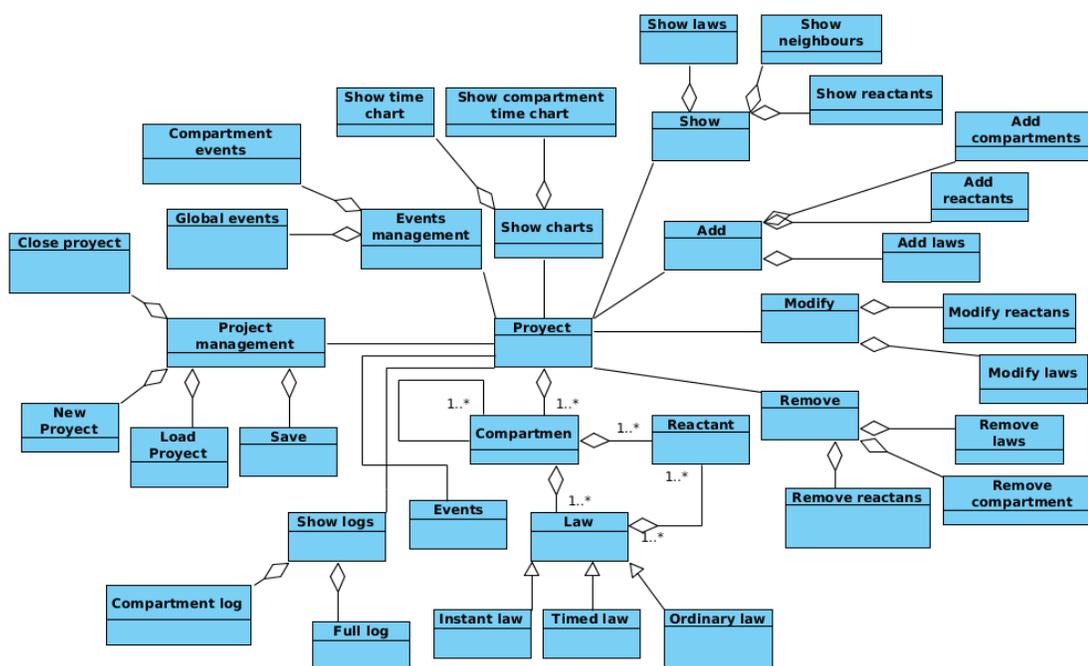
**Figura 7.16** Diagrama de clases que ilustra las principales clases de la lógica del Sistema de Recuperación de Proyectos de Software RPS-GS. Vista B.



**Figura 7.17** Diagrama de clases correspondiente a la funcionalidad “Añadir Reacción Química”, de la plataforma bioinformática Cellulat.



**Figura 7.18** Diagrama de clases correspondiente a la funcionalidad “Modificar Reacción Química”, de la plataforma bioinformática Cellulat.



**Figura 7.19** Diagrama de clases que ilustra las principales clases de la lógica de la plataforma bioinformática Cellulat.

## VII.2.3 Diagramas de interacción

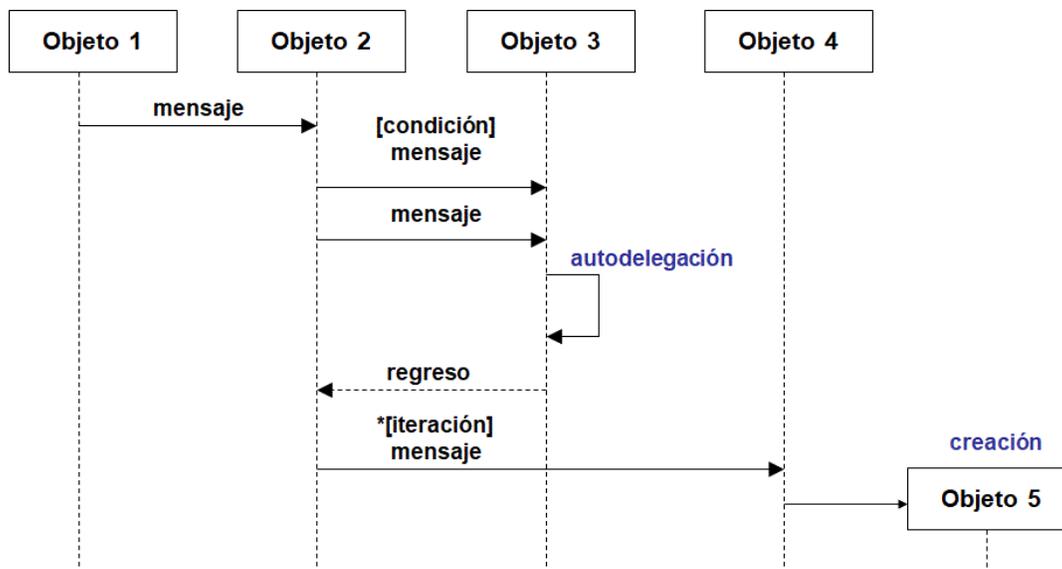
### VII.2.3.1 Diagramas de secuencia

Los diagramas de interacción [6,7] –diagramas de secuencia y diagramas de colaboración– modelan el comportamiento del sistema a través de la interacción entre los objetos que lo conforman. Describen las secuencias de paso de mensajes entre los objetos que implementan el comportamiento del sistema. Los componentes del diagrama de secuencia son los objetos, la línea de vida de los objetos y los

mensajes. Los componentes del diagrama de colaboración son los objetos, los enlaces y los mensajes.

Un diagrama de secuencia representa la interacción que ocurre entre objetos a través del tiempo. Como se puede apreciar en la figura 7.20, tres símbolos básicos son usados en un diagrama de secuencia:

- Rectángulos, que representan objetos.
- Líneas verticales discontinuas, cada una de las cuales inicia en la base del rectángulo y representa el tiempo de vida del objeto (línea de vida del objeto) durante la interacción.
- Flechas horizontales, que representan mensajes y van desde una línea de vida de un objeto a la línea de vida de otro objeto.



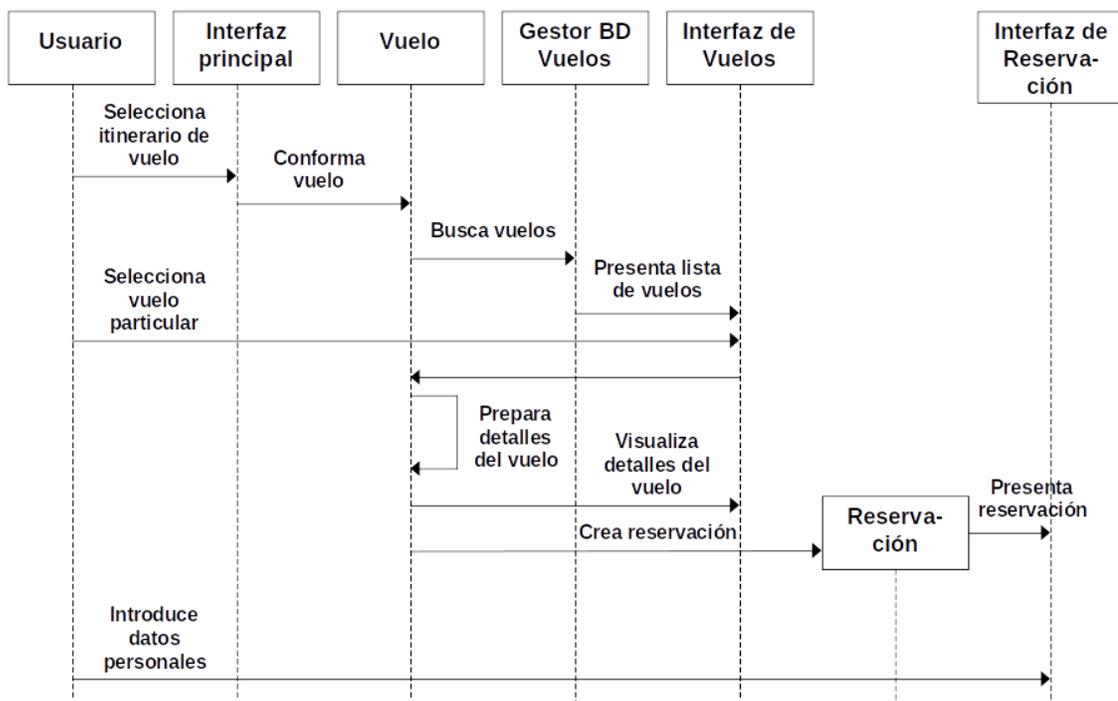
**Figura 7.20** Componentes de un diagrama de secuencia.

El orden en el que se dan los mensajes es de arriba hacia abajo. A cada mensaje se asocia una etiqueta la cual corresponde al nombre del mensaje. También se pueden incluir los argumentos e información de control en la etiqueta. La información de control en la etiqueta contiene dos partes fundamentales: una condición y un marcador de iteración. La condición aparece entre corchetes ([condición]) e indica cuando un mensaje debe ser enviado. El marcador de iteración muestra que un mensaje se envía varias veces a diferentes objetos receptores. La notación del marcador de iteración es un juego de corchetes precedido por un asterisco (\*[ ]).

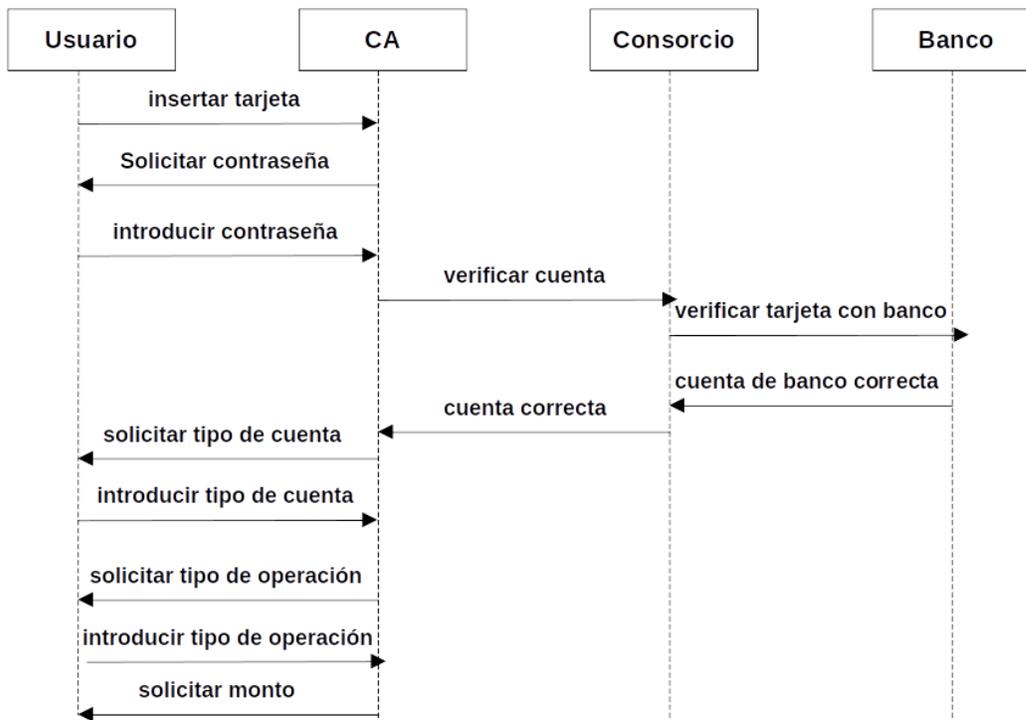
Como se puede apreciar en la Figura 7.20, una flecha con línea discontinua en sentido contrario indica el regreso de un mensaje y no un nuevo mensaje. En dicho

diagrama también se muestra un ejemplo de auto delegación, que es un mensaje que un objeto se envía a sí mismo. En este caso, la flecha regresa a la misma línea de vida de donde partió.

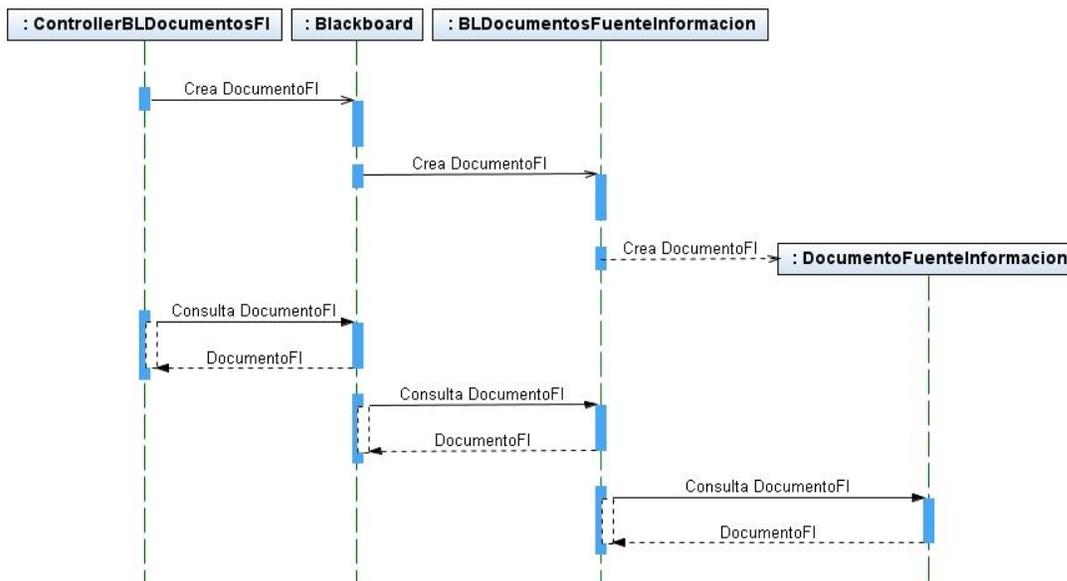
Las Figuras 7.21 a la 7.24 ilustran el uso de los diagramas de secuencia durante el diseño detallado de funcionalidades de cuatro sistemas de software. La Figura 7.21 muestra el diagrama de secuencia que modela la funcionalidad “*Efectuar reservación de vuelo*”, del sistema Web de reservaciones y ventas de una aerolínea. En la Figura 7.22, se ilustra el diagrama de secuencia correspondiente a la funcionalidad “*Validación de datos y solicitud de tipo de operación*” del software de un cajero automático. El diagrama de secuencia que modela la funcionalidad “*Gestión de documentos del proyecto de software*” el sistema RPS-GS se muestra en la Figura 7.23. Finalmente, en la Figura 7.24 se ilustra el diagrama de secuencia que modela la funcionalidad “*Añadir reacción química*” de la plataforma bioinformática Cellulat.



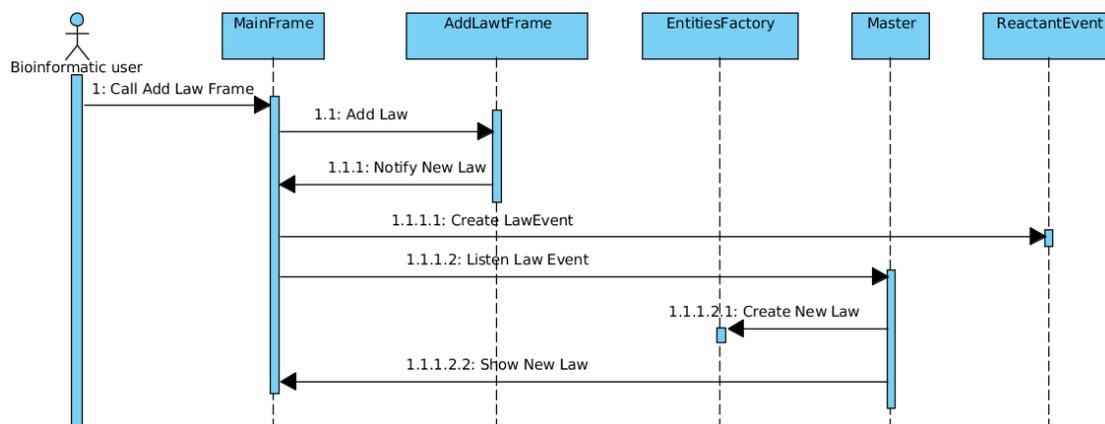
**Figura 7.21** Diagrama de secuencia correspondiente a la funcionalidad “Efectuar reservación de vuelo”, del sistema Web de reservaciones y ventas de una aerolínea.



**Figura 7.22** Diagrama de secuencia correspondiente a la funcionalidad “Validación de datos y solicitud de tipo de operación”, del sistema de un cajero automático.



**Figura 7.23** Diagrama de secuencia correspondiente a la funcionalidad “Gestión de documentos del proyecto de software”, del sistema RPS-GS.



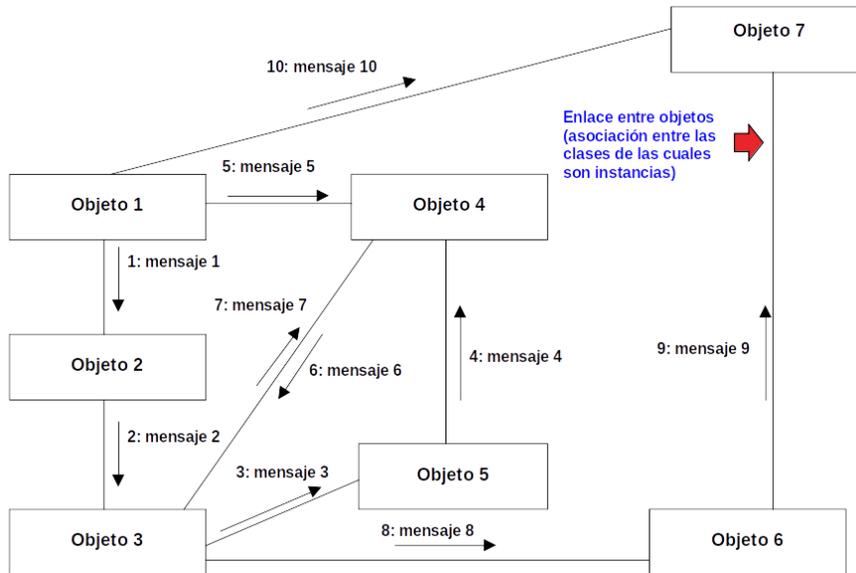
**Figura 7.24** Diagrama de secuencia correspondiente a la funcionalidad “Añadir reacción química”, de la plataforma bioinformática Cellulat.

### VII.2.3.2 Diagramas de colaboración

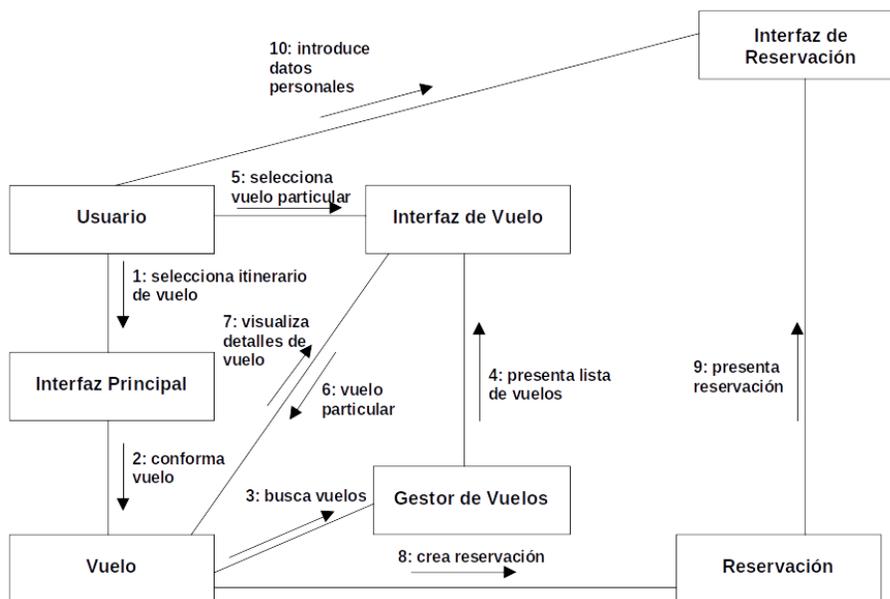
Por otra parte, al igual que un diagrama de secuencia, el diagrama de colaboración [6,7] representa la interacción que ocurre entre los objetos de un caso de uso, pero sin mostrar explícitamente la secuencia de la interacción en el tiempo. A diferencia de un diagrama de secuencia, el cual se define en términos de tres símbolos básicos (objeto, línea de vida y mensaje), el diagrama de colaboración no incluye las líneas de vida y define un nuevo símbolo: los enlaces entre objetos, representados por líneas (como se representan las asociaciones en un diagrama de clases).

Otra diferencia entre el diagrama de secuencias y el diagrama de colaboración, consiste en que el último enumera los mensajes que se pasan entre los objetos. Cada mensaje se representa mediante una flecha etiquetada unida a la línea del enlace. A diferencia del diagrama de secuencias, el diagrama de colaboración muestra cómo se vinculan entre sí los objetos al hacer visibles todos los enlaces existentes entre éstos. Una ventaja de los diagramas de colaboración es la claridad con la que muestran las relaciones entre los objetos. Sin embargo, la secuencia de tiempo se debe obtener a partir de los números de los mensajes.

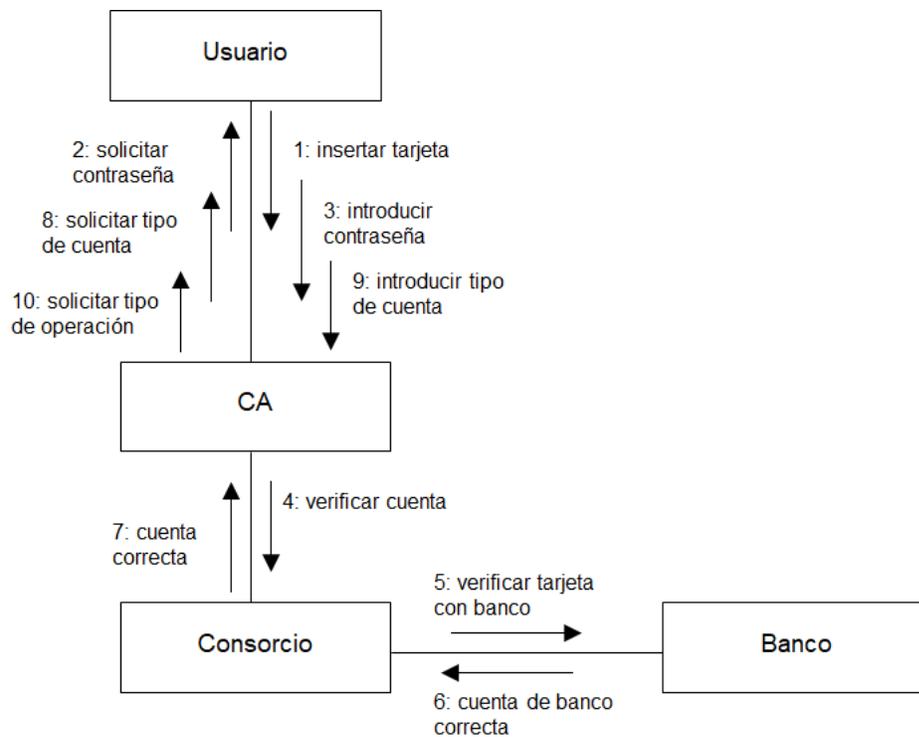
La Figura 7.25 muestra una representación genérica de un diagrama de colaboración. Por otra parte, las Figuras 7.26 y 7.27 ilustran el uso de los diagramas de colaboración en dos escenarios ya introducidos.



**Figura 7.25** Representación genérica de un diagrama de colaboración.



**Figura 7.26** Diagrama de colaboración correspondiente a la funcionalidad "Interacción con la interfaz gráfica de Reservas", del sistema Web de reservas y ventas de una aerolínea.



**Figura 7.27** Diagrama de colaboración correspondiente a la funcionalidad “Validación de sats y solicitud de tipo de operación”, del sistema de un cajero automático.

#### VII.2.4 Diagramas de transición de estados

Un diagrama de estados [6,7] describe todos los estados posibles por los que puede transitar un objeto particular, como resultado de los eventos que llegan a éste. Éste relaciona eventos y estados. Cuando se recibe un evento, el estado siguiente depende del actual, así como del evento recibido. Un cambio de estado causado por un evento es lo que se conoce como transición. La representación de un diagrama de estados es un grafo cuyos nodos son estados, y cuyos arcos dirigidos son transiciones etiquetadas con nombres de eventos. Una transición que sale de un estado define la respuesta del objeto en ese estado a la ocurrencia de un evento.

Los estados se representan como rectángulos redondeados que contienen un nombre. Dicho nombre se refiere al estado actual en el que se encuentra el objeto. Las transiciones se representan en forma de flechas desde el estado receptor hasta el estado destino. La etiqueta de la flecha es el nombre del evento que da lugar a la transición. Una transición se caracteriza por los siguientes elementos: evento que la activa, condición de guarda (expresión booleana), una acción y un estado destino. Todas las transiciones que salgan de un estado deben de corresponder a eventos distintos. El diagrama de estados especifica la secuencia de estados que causa una cierta secuencia de eventos. La Figura 7.28 muestra una representación genérica de un diagrama de estados. Por su parte, las Figuras 7.29 y 7.30 ilustran los diagramas de estados de objetos pertenecientes a las aplicaciones antes vistas.

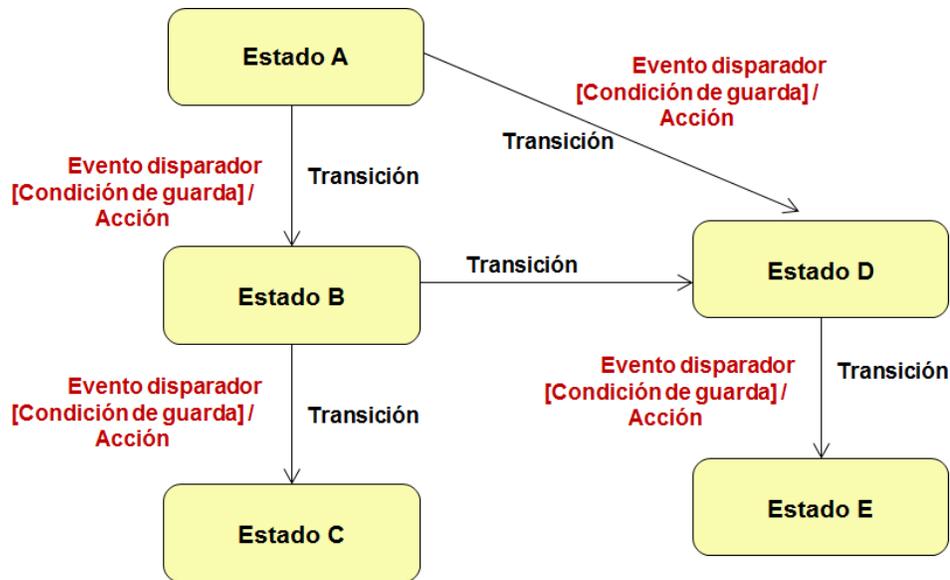


Figura 7.28 Representación genérica de un diagrama de estados.

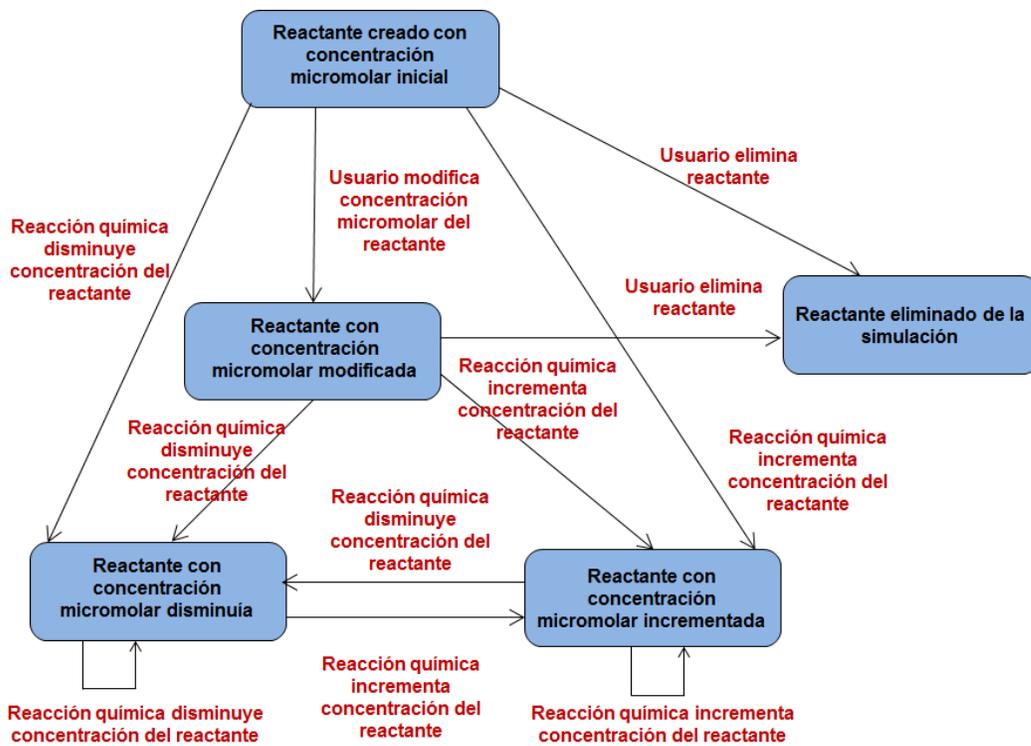
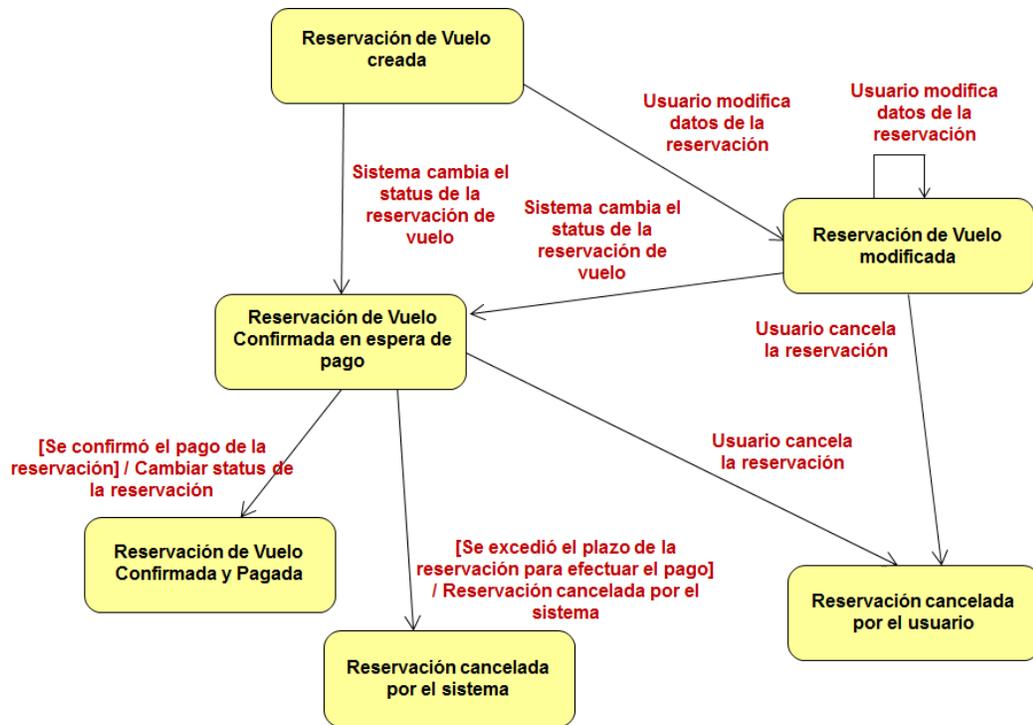


Figura 7.29 Diagrama de estados del objeto "Reactante", de la plataforma bioinformática Cellulat.

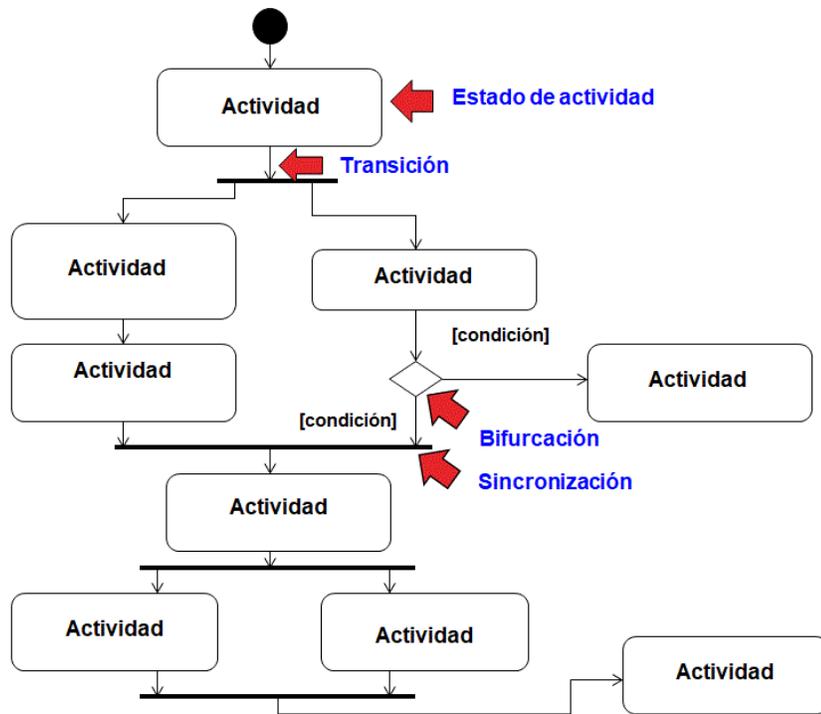


**Figura 7.30** Diagrama de estados del objeto “Reservación de Vuelo”, del sistema Web de reservas y ventas de una aerolínea.

### VII.3 Diagramas de Actividades

El diagrama de actividades [6,7] modela los estados de ejecución del cómputo (la ejecución del sistema) y el flujo de trabajo. El diagrama de actividades posee algunas semejanzas con el diagrama de flujo de datos. La principal diferencia entre ambos diagramas reside en que los diagramas de flujo de datos se limitan comúnmente a procesos secuenciales, mientras que los diagramas de actividades pueden manejar además procesos concurrentes y paralelos. En un diagrama de actividades, el modelado de los hilos concurrentes, se utiliza para representar actividades que se pueden realizar concurrentemente por los diferentes objetos.

A diferencia del diagrama de estados, que muestra la transición por los diferentes estados que puede alcanzar un objeto particular, el diagrama de actividades modela los estados de ejecución del cómputo (la ejecución del sistema) y el flujo de trabajo.



**Figura 7.31** Representación genérica de un diagrama de actividades.

Como se ilustra en la Figura 7.31, un diagrama de actividades incluye los siguientes símbolos: rectángulos de bordes redondeados para representar estados de actividad, flechas para representar transiciones simples de terminación, rombos para representar bifurcaciones, y barras gruesas para representar sincronización.

**Estado de actividad.** La interpretación del término estado de actividad depende de la perspectiva o nivel de detalle que ofrezca el diagrama de actividades. Considerando lo anterior, podemos decir que un estado de actividad puede representar desde una determinada tarea que debe ser ejecutada ya sea por un ser humano o por la computadora, hasta la ejecución de un método de una clase.

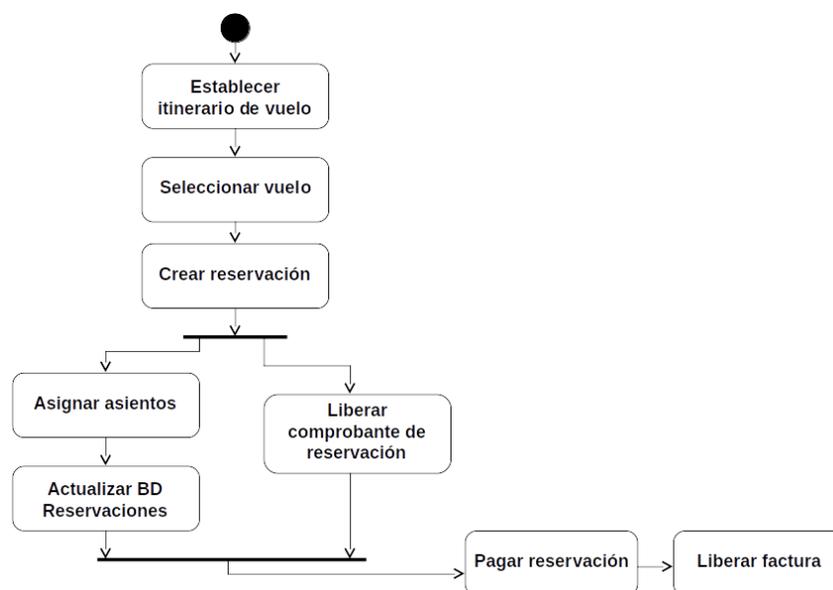
**Transiciones simples.** Representa el fin de una actividad y el paso de la ejecución al siguiente estado de actividad en el diagrama. A diferencia de un diagrama de estados, donde la transición de un estado a otro se da a partir de la ocurrencia de un evento, en el diagrama de actividades la transición ocurre cuando un estado de actividad ha finalizado la ejecución de su cómputo.

**Bifurcaciones.** Establecen las dos diferentes ramas que se pueden seguir después de evaluar una condición booleana.

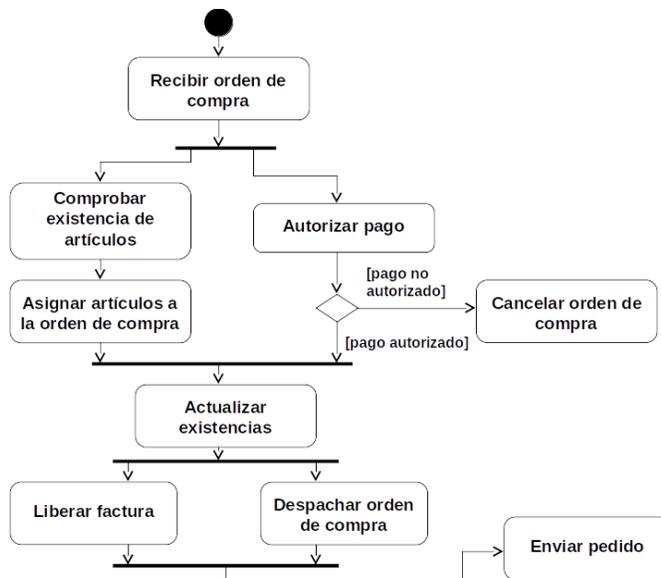
**Sincronización.** Corresponde a una división o unión de control, la cual se representa por una barra gruesa con varias flechas que llegan o salen. Cada flecha que llega a la barra de sincronización corresponde al final de la ejecución de un hilo concurrente.

Otra característica relevante de los diagramas de actividades es su capacidad de mostrar comportamientos que abarcan varios casos de uso. En este sentido, los diagramas de actividades nos proporcionan valiosa información sobre el comportamiento de dos o más casos de uso interconectados. Los diagramas de actividades muestran el flujo de actividades, pero no los objetos que realizan dichas actividades. Podemos considerar que los diagramas de actividades son el punto de partida para el diseño, dado que nos muestran las principales actividades que el sistema debe ejecutar. El trabajo de diseño continúa en la medida en que las actividades se descomponen en operaciones y estas operaciones son asignadas a clases específicas para su implementación.

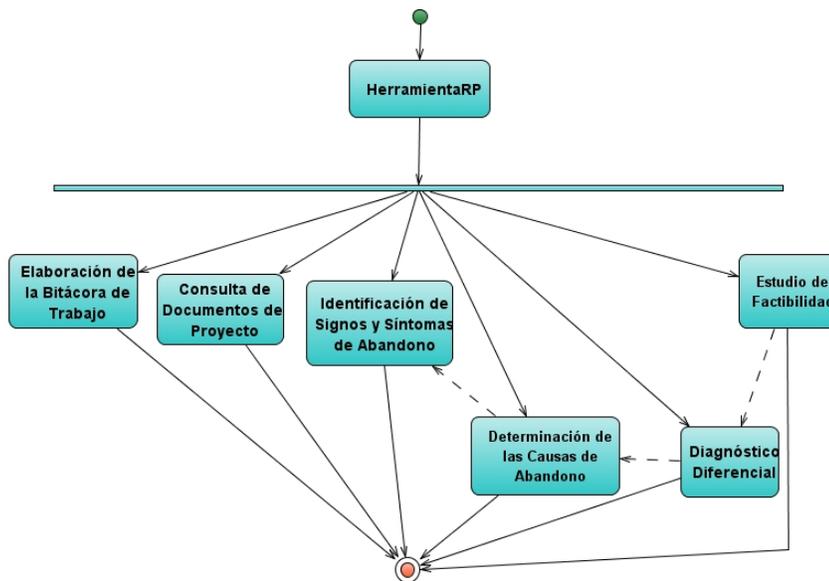
Las Figuras 7.32 a 7.34 ilustran el uso de los diagramas de actividades durante la fase de diseño detallado del sistema Web de reservas y ventas de una aerolínea, un sistema de control de pedidos, y del sistema RPS-GS, respectivamente.



**Figura 7.32** Diagrama de actividades correspondiente a la funcionalidad "Efectuar reservación", del sistema Web de reservas y ventas de una aerolínea.



**Figura 7.33** Diagrama de actividades correspondiente a la funcionalidad “Gestión de órdenes de compra”, de un sistema de control de pedidos.



**Figura 7.34** Diagrama de actividades a nivel sistema, del sistema RPS-GS.

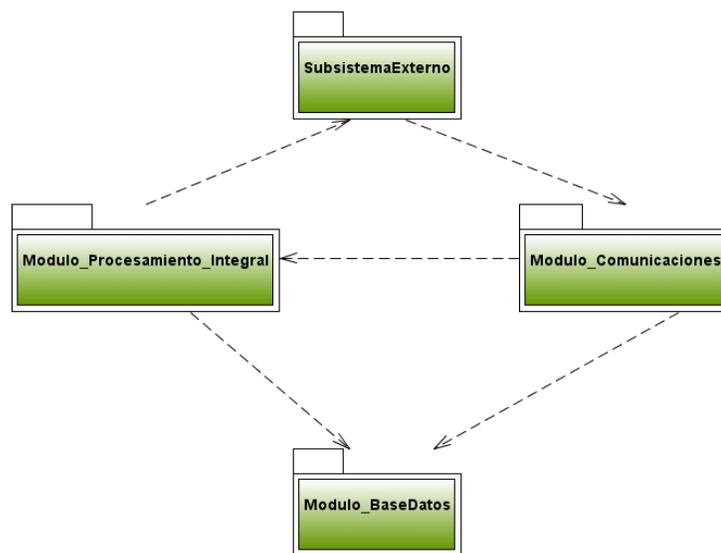
## VII.4 Casos de estudio

En el epígrafe V.9 ilustramos algunos aspectos del diseño arquitectónico del subsistema SPI y de la plataforma bioinformática Evolution. En ambos casos vimos que ambas arquitecturas se basan en la integración de los modelos arquitectura de

pizarra y MVC en una misma solución arquitectónica. En este epígrafe retomaremos estos dos casos de estudio, e ilustraremos algunas de las decisiones tomadas en la modularización y el diseño de componentes durante el desarrollo de cada uno de estos sistemas.

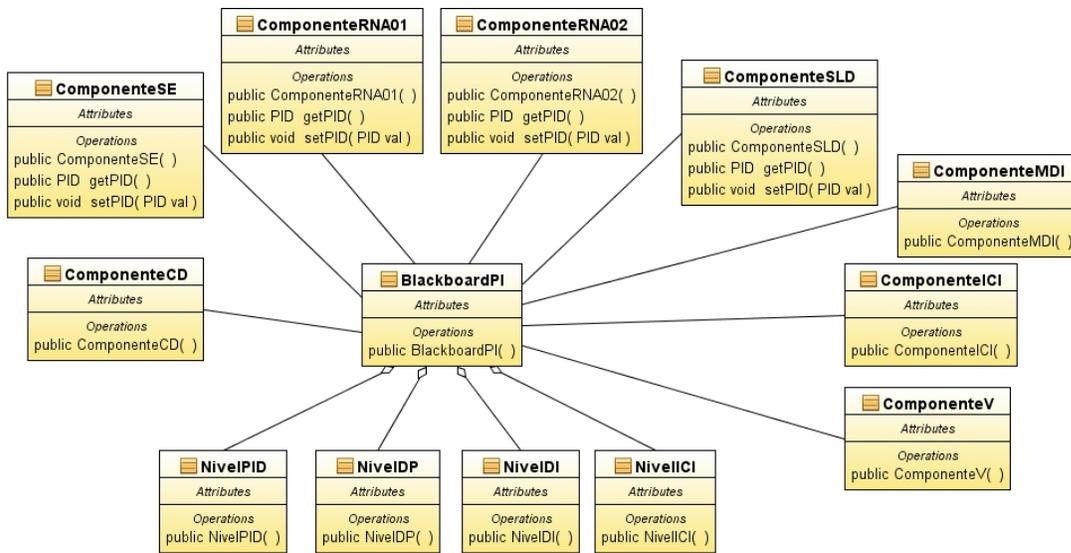
#### VII.4.1 Sistema de detección de fugas en ductos que transportan hidrocarburos. modularización y diseño de componentes

Dada la gran complejidad que caracteriza al subsistema SPI, en términos de los módulos, componentes, clases y relaciones que lo integran, aquí solo nos concentraremos en presentar algunos de los aspectos del diseño detallado del Módulo de Procesamiento Integral (ver Figura 7.35). En la Figura 7.35 se ilustra el primer nivel de modularización propuesto. Como se puede apreciar en esta figura, el subsistema SPI organiza su estructura en términos de tres módulos: Módulo de Procesamiento Integral, Módulo de Comunicaciones, Módulo de Bases de Datos. El módulo denominado Subsistema Externo representa sistemas remotos con los cuales el subsistema SPI interactúa. Tanto el Módulo de Procesamiento Integral como el Módulo de Comunicaciones establecen relaciones de dependencia con el Módulo Base de Datos. Por su parte, el Módulo de Comunicaciones también establece una relación de dependencia con el Módulo de Procesamiento Integral.

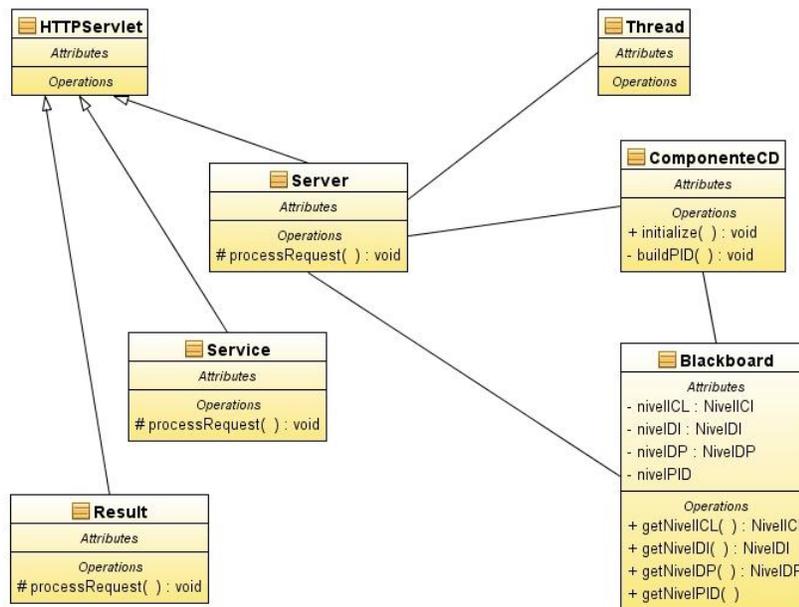


**Figura 7.35** Primer nivel de modularización propuesto para el subsistema SPI.

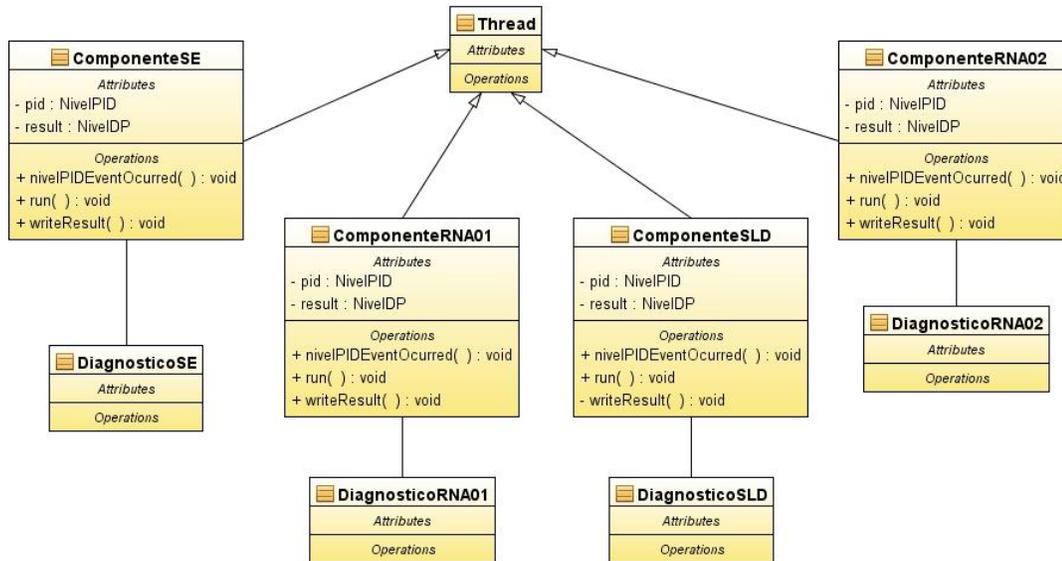
Siguiendo con el diseño del Módulo de Procesamiento Integral, las Figuras 7.36 a la 7.38 muestran aspectos claves de la vista estática (diagramas de clases) del *blackboard*, los niveles que lo integran, y de los componentes de diagnóstico que operan sobre éste.



**Figura 7.36** Modularización y diseño detallado (diagrama de clases) del BlackboardPI, los niveles que lo integran, así como los componentes que interactúan con éste.

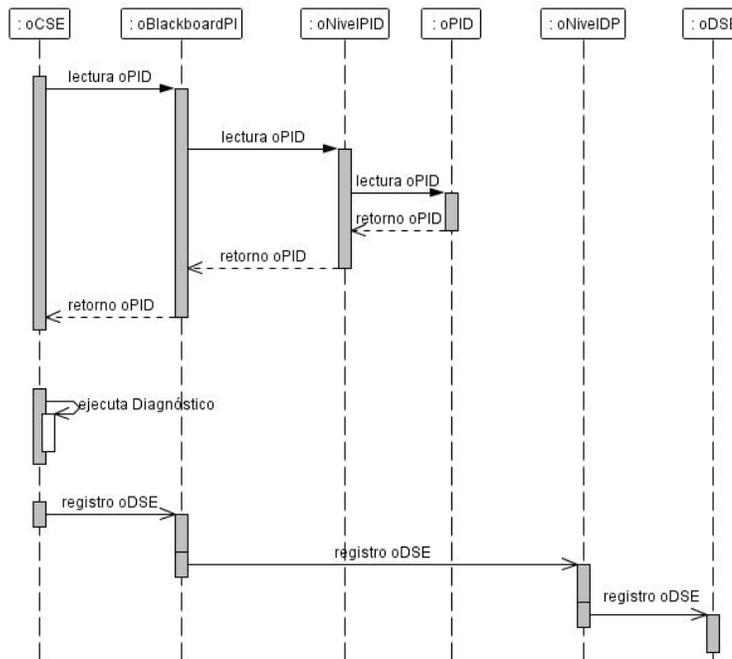


**Figura 7.37** Diseño detallado (diagrama de clases) que ilustra la relación de las clases Server, ComponenteCD y Blackboard.

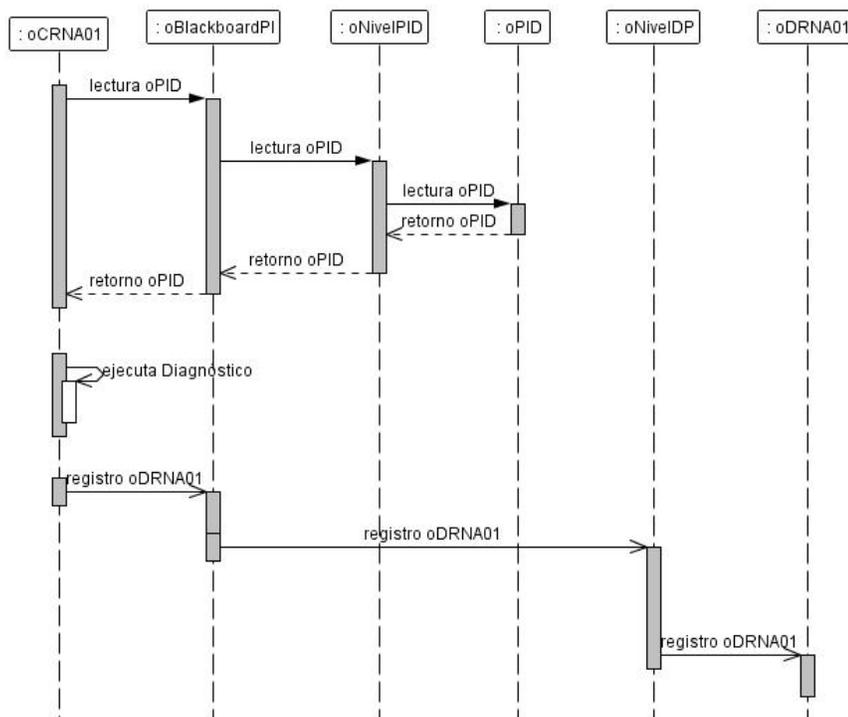


**Figura 7.38** Diseño detallado (diagrama de clases) que ilustra todos los componentes de diagnóstico como tipos de Thread.

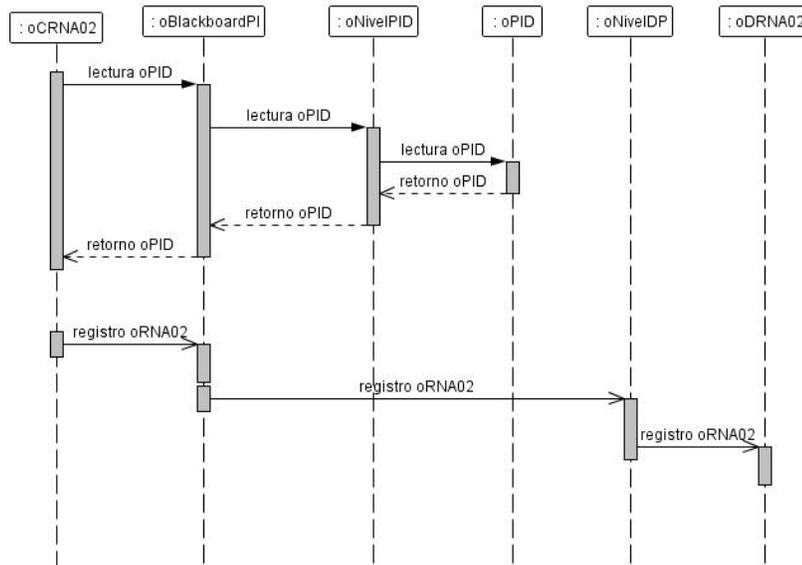
Las Figuras 7.39 a la 7.44 ilustran, a través de diagramas de secuencia, como se lleva a cabo la interacción entre los componentes que operan sobre el BlackboardPI (ver Figura 7.36) y los niveles de este último, involucrados en la lectura y creación de elementos solución por parte de dichos componentes.



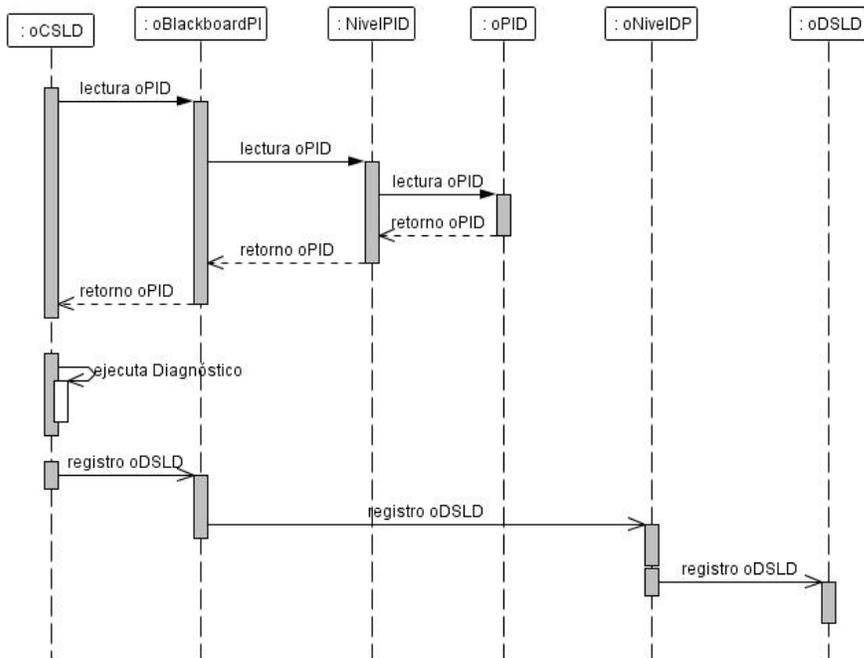
**Figura 7.39** Diseño detallado (diagrama de secuencias) que ilustra la ejecución del diagnóstico particular por el ComponenteSE (oCSE).



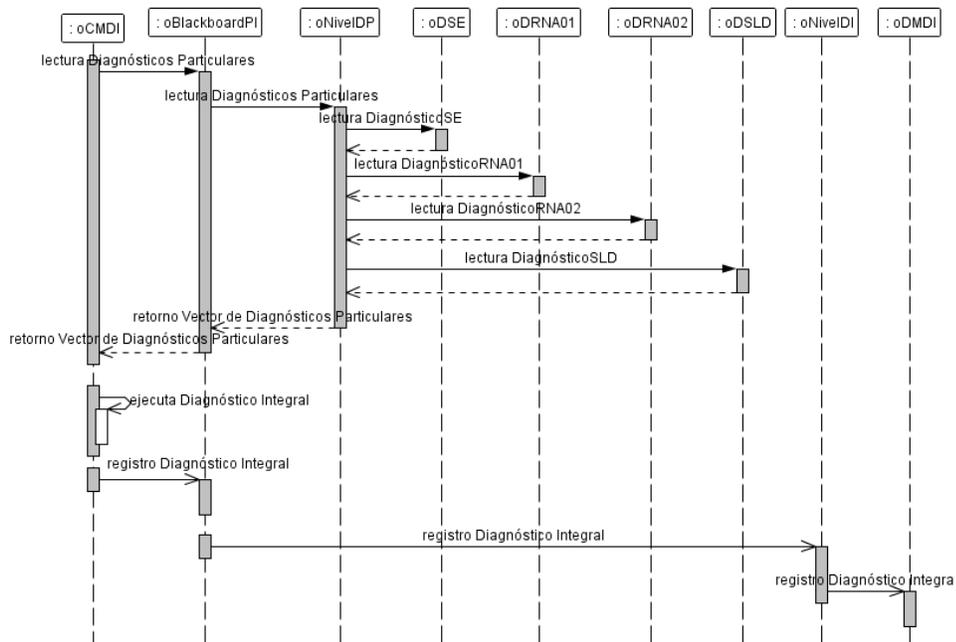
**Figura 7.40** Diseño detallado (diagrama de secuencias) que ilustra la ejecución del diagnóstico particular por el ComponenteRNA01 (oCRNA01).



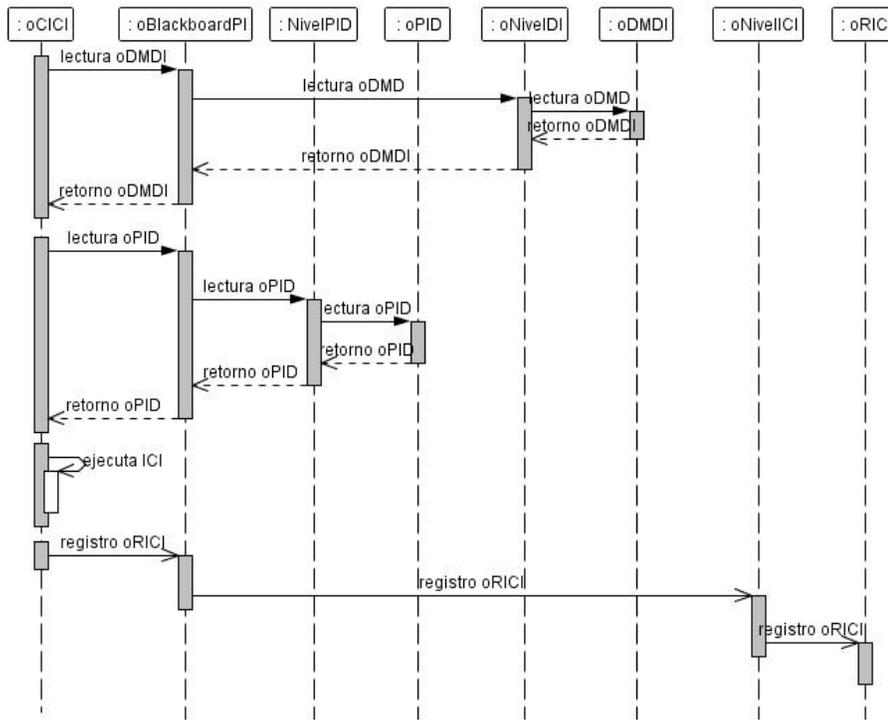
**Figura 7.41** Diseño detallado (diagrama de secuencias) que ilustra la ejecución del diagnóstico particular por el ComponenteRNA02 (oCRNA02).



**Figura 7.42** Diseño detallado (diagrama de secuencias) que ilustra la ejecución del diagnóstico particular por el ComponenteSLD (oCSLD).

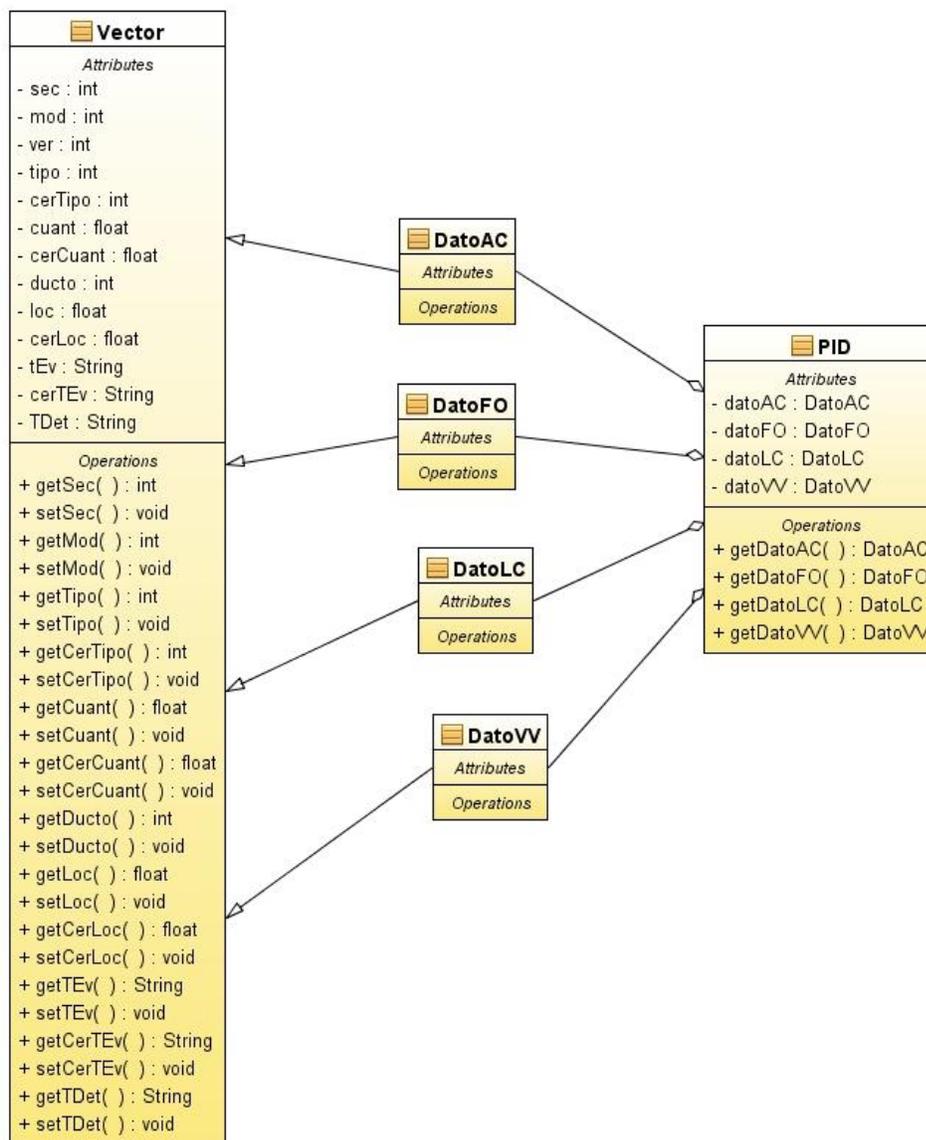


**Figura 7.43** Diseño detallado (diagrama de secuencias) que ilustra la ejecución del diagnóstico integral por el ComponenteCMDI (oCMDI).



**Figura 7.44** Diseño detallado (diagrama de secuencias) que ilustra la ejecución del completamiento e integración de la información por el ComponenteICI (oCICI).

Las Figuras 7.45 a la 7.52 ilustran, a través de diagramas de clases, parte el diseño detallado relacionado con la identificación de nuevas relaciones, así como atributos y métodos del patrón integral de datos (PID), del *blackboard*, así como de los componentes que operan sobre el *blackboard*.



**Figura 7.45** Diseño detallado (diagrama de clase) que ilustra el patrón integral de datos (PID) a detalle.

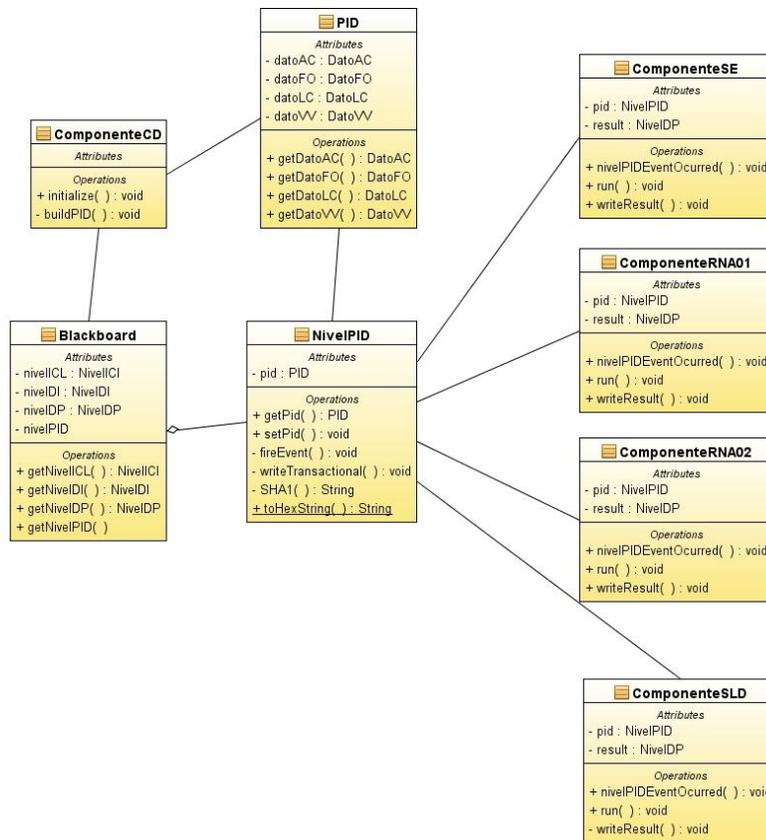


Figura 7.46 Diseño detallado (diagrama de clase) que ilustra los niveles del Blackboard como contenedores de objetos y su relación con los componentes de diagnóstico.

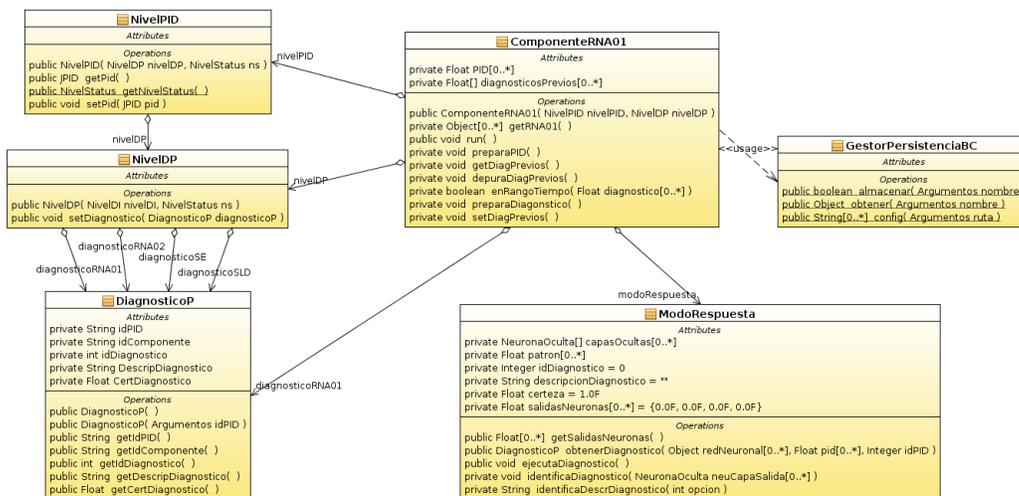


Figura 7.47 Diseño detallado (diagrama de clase) que ilustra la interacción entre el ComponenteRNA01 y el Blackboard.

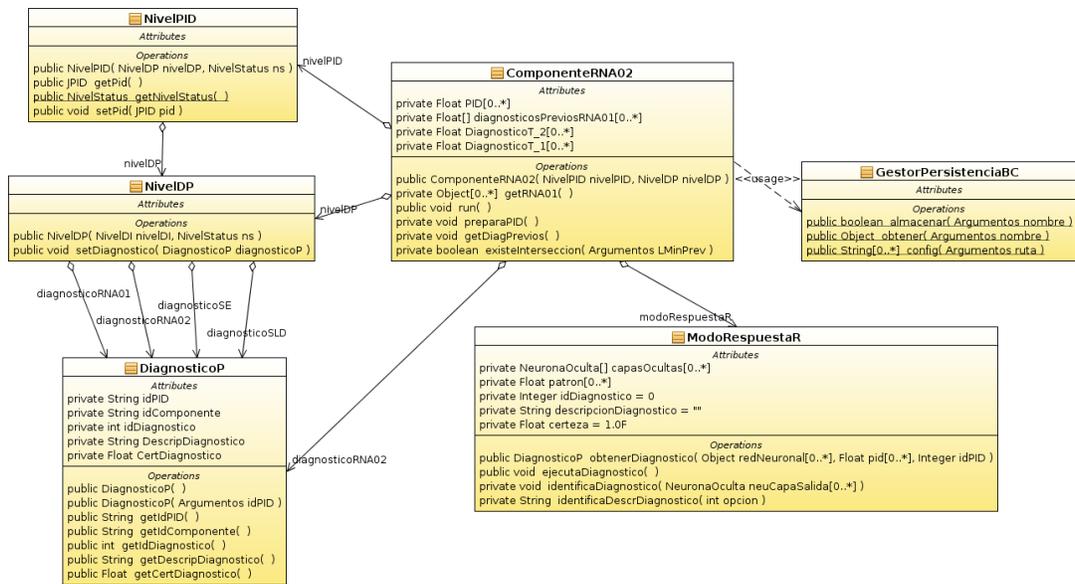


Figura 7.48 Diseño detallado (diagrama de clase) que ilustra la interacción entre el ComponenteRNA02 y el Blackboard.

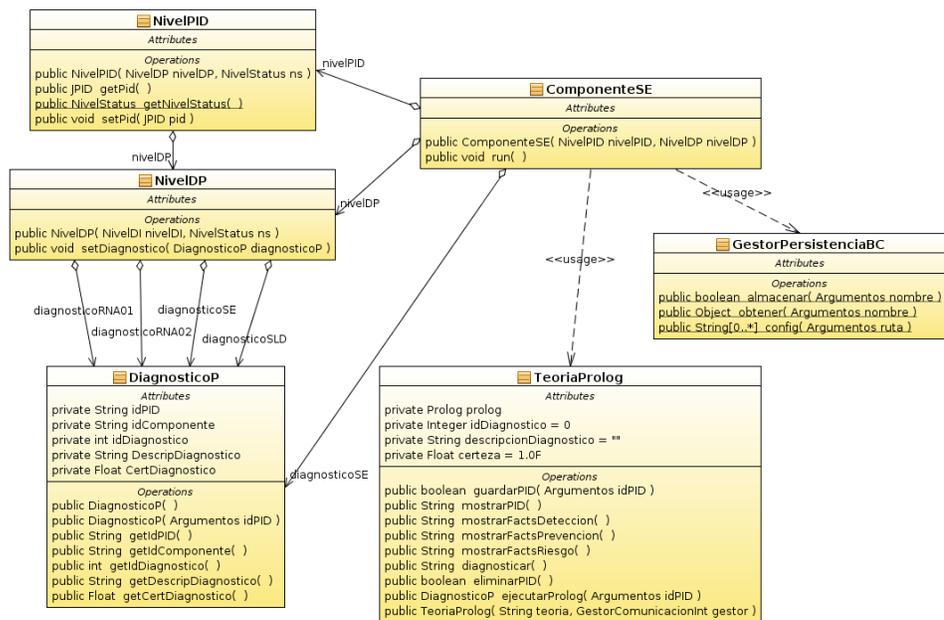
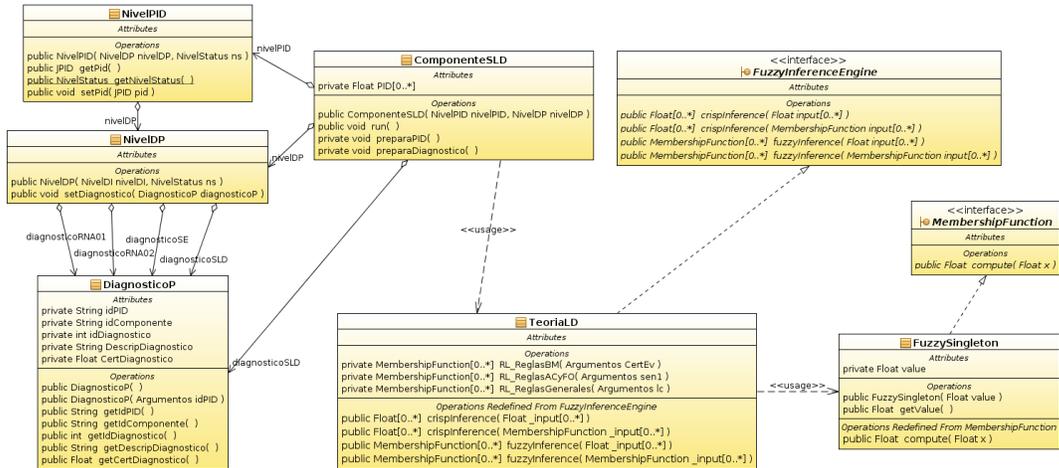
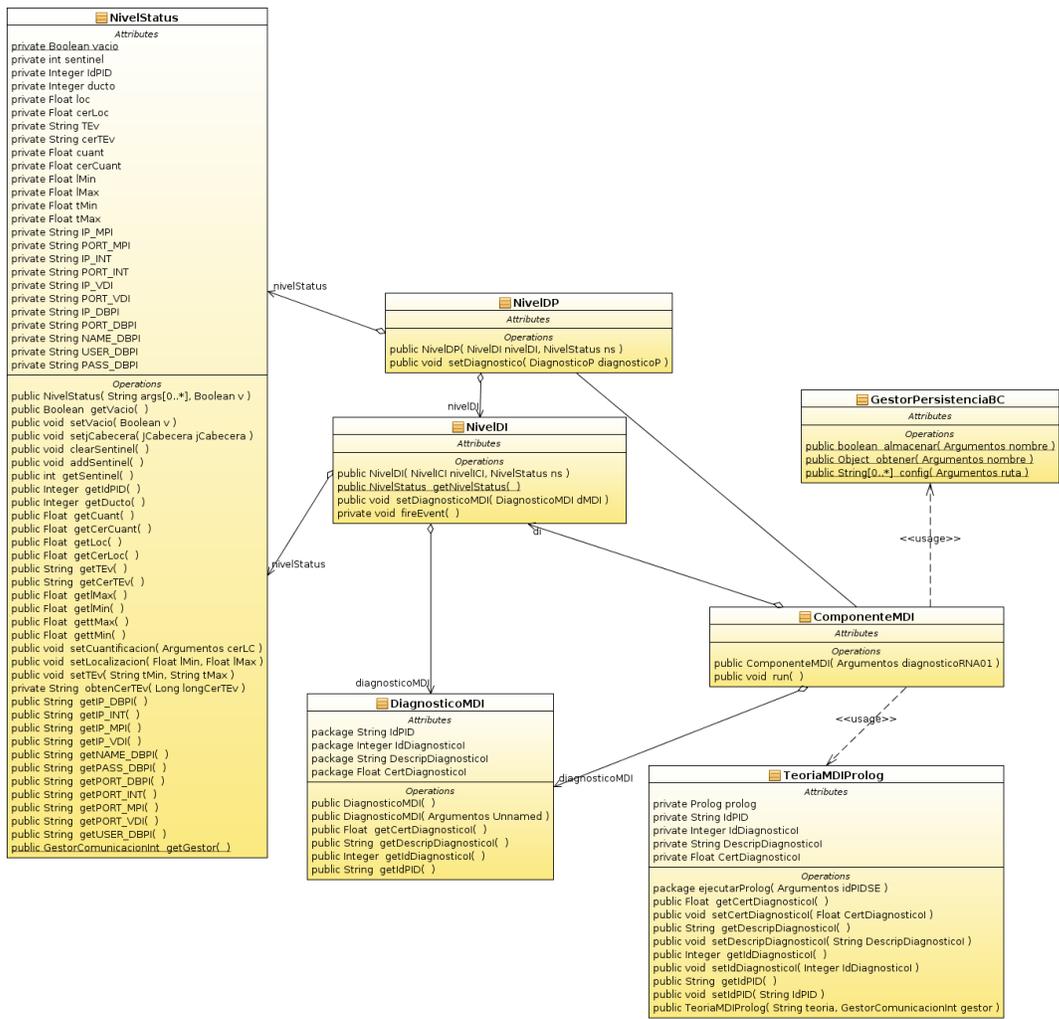


Figura 7.49 Diseño detallado (diagrama de clase) que ilustra la interacción entre el ComponenteSE y el Blackboard.



**Figura 7.50** Diseño detallado (diagrama de clase) que ilustra la interacción entre el ComponenteSLD y el Blackboard.



**Figura 7.51** Diseño detallado (diagrama de clase) que ilustra la interacción entre el ComponenteMDI y el Blackboard.

Modularización y Diseño de Componentes  
Casos de estudio



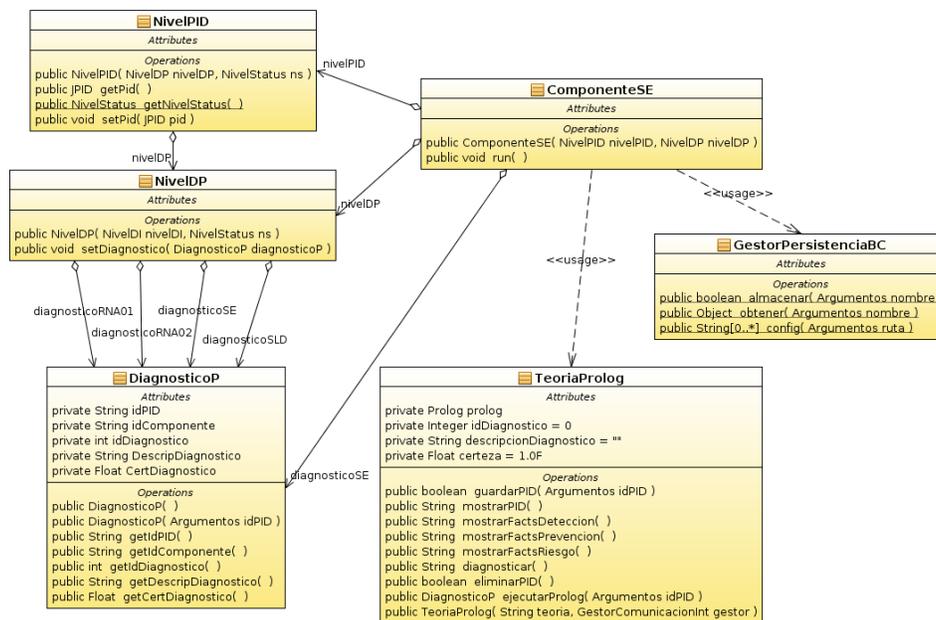


Figura 7.54 Diseño detallado final (diagrama de clase) del componente de diagnóstico ComponenteSE.

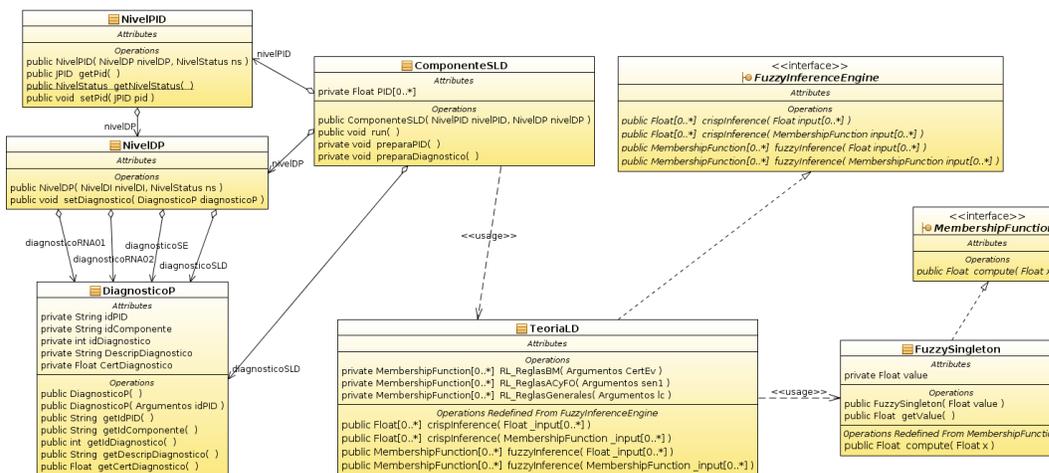
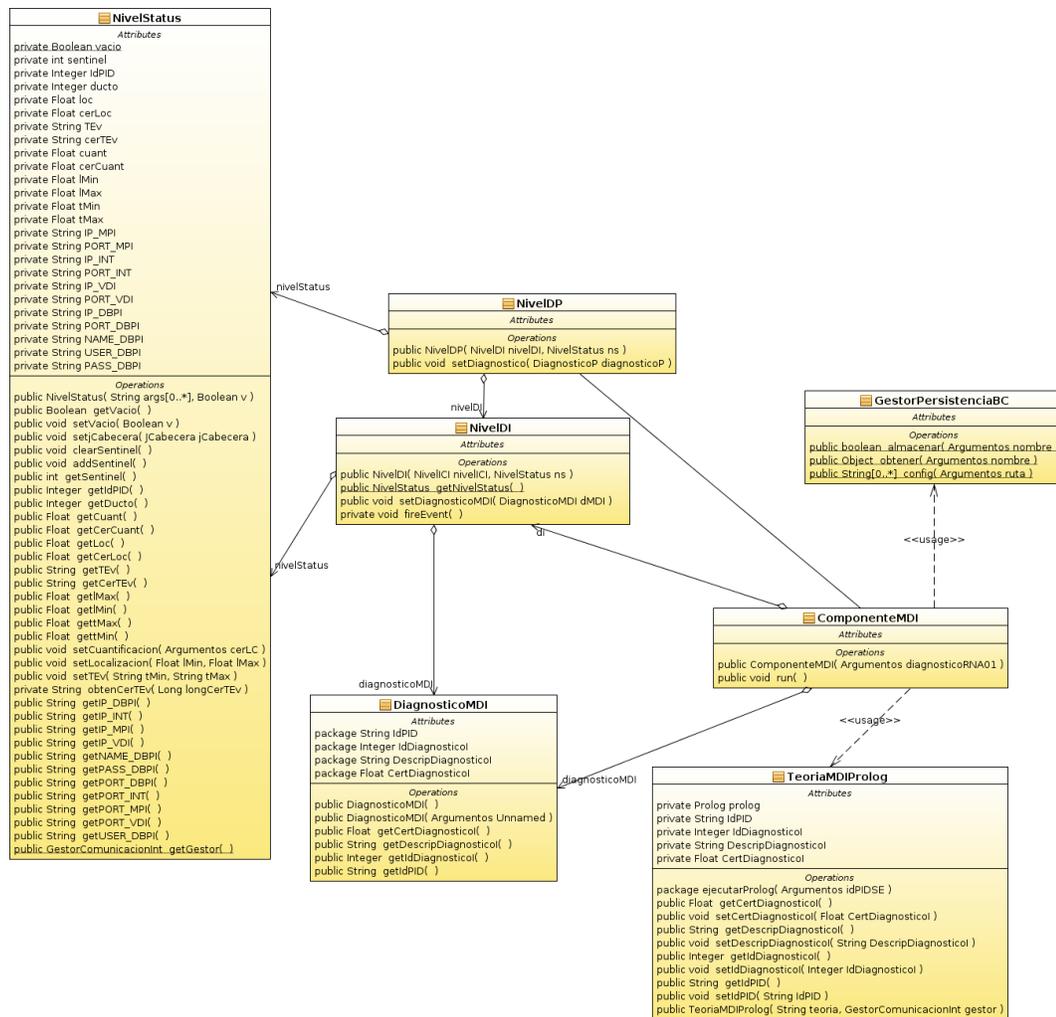
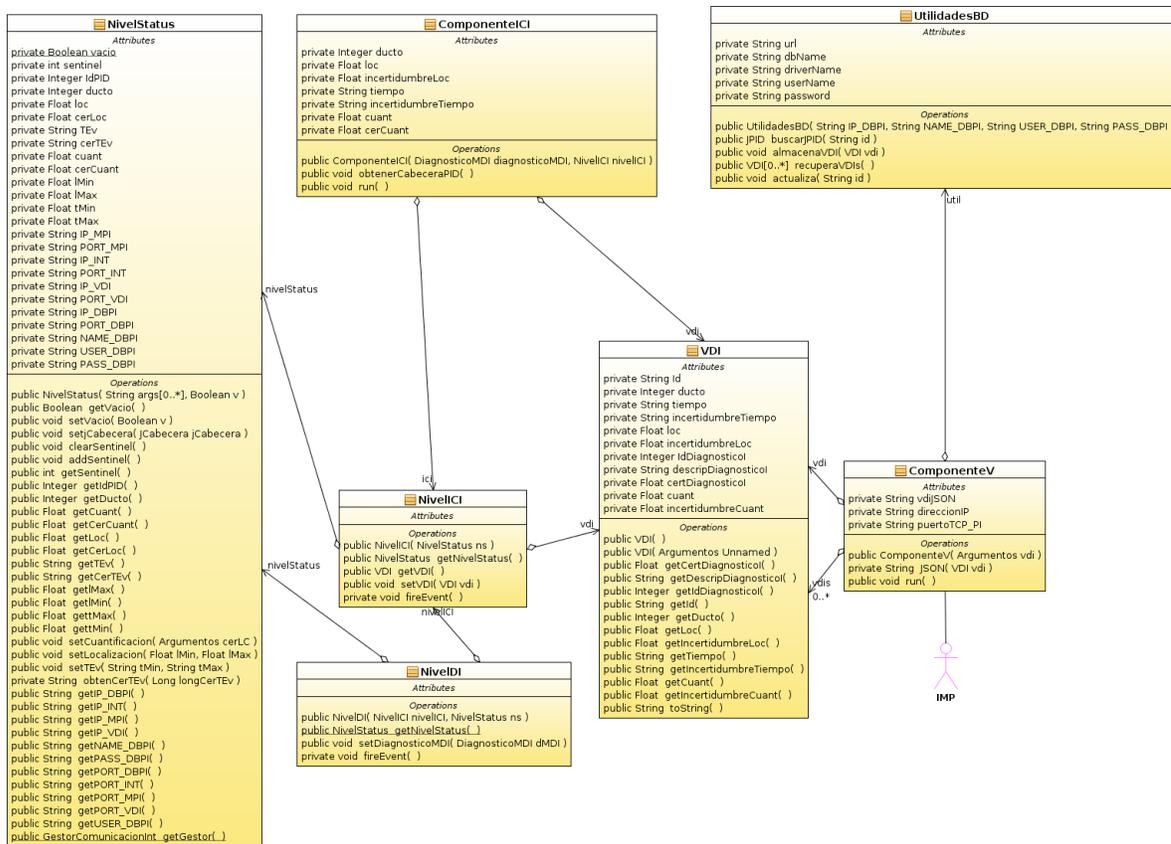


Figura 7.55 Diseño detallado final (diagrama de clase) del componente de diagnóstico particular ComponenteSLD.



**Figura 7.56** Diseño detallado final (diagrama de clase) del componente de diagnóstico integral ComponenteMDI.

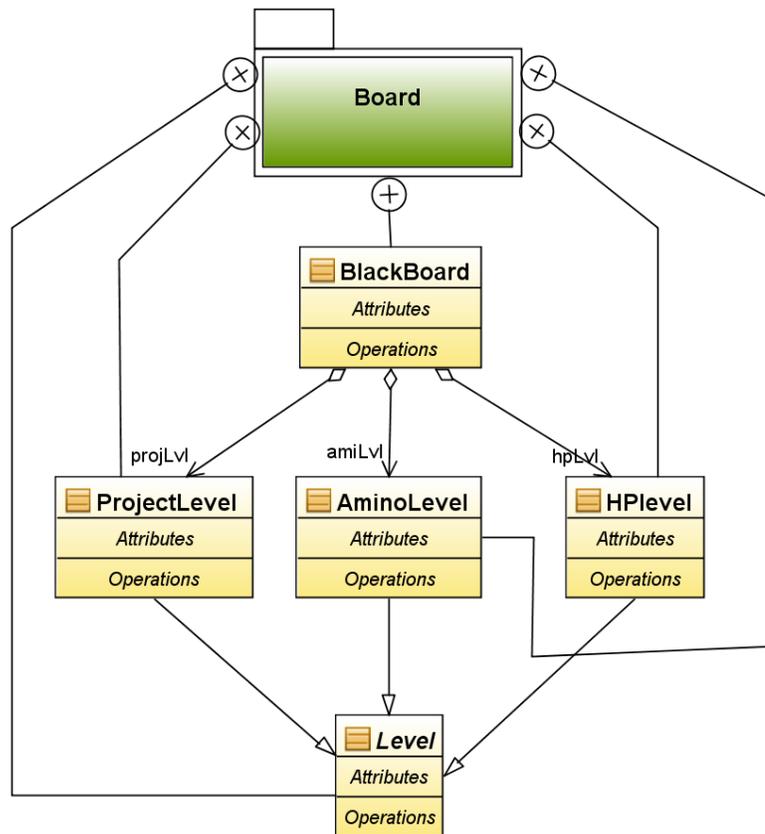


**Figura 7.57** Diseño detallado final (diagrama de clase) de los componentes de integración y completamiento de la información (ComponenteICI) y de visualización de la información (ComponenteV).

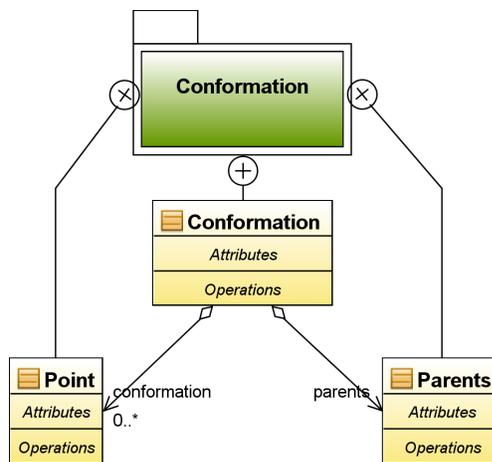
### VII.4.2 Plataforma bioinformática evolution. modularización y diseño detallado.

En el epígrafe V.9.2 nos referimos a la arquitectura lógica exhibida por la plataforma bioinformática Evolution, en términos de las tres capas que la conforman –Vista, Controlador y Modelo– así como al primer nivel de modularización propuesto, es decir, el nivel de modularización a nivel superior que proporciona una vista macro de los componentes que integran cada una de las tres capas. En este epígrafe, tanto para ilustrar la subsecuente modularización como parte del nivel detallado, nos referiremos solo a aquellos aspectos del diseño con el blackboard, las conformaciones, las generaciones, así como con el algoritmo evolutivo y los diferentes tipos de operadores genéticos encargados del proceso de optimización de las conformaciones generadas inicialmente de forma aleatoria sobre el blackboard.

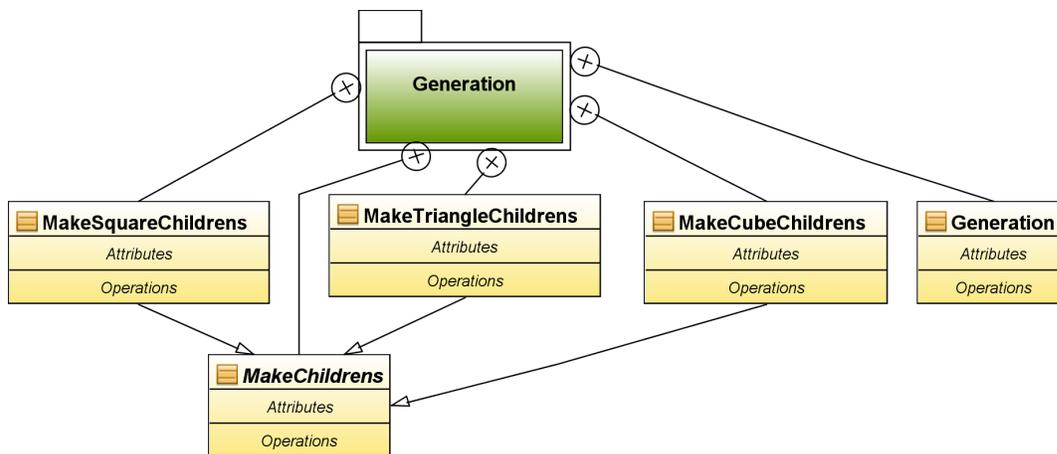
Las Figuras 7.58 a la 7.65 ilustran tanto el segundo nivel de modularización adoptado como parte del diseño detallado de las principales clases contenidas en cada uno de los módulos.



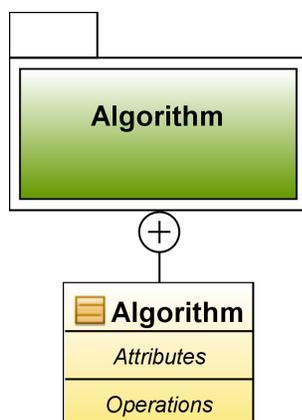
**Figura 7.58** El módulo *Board* (*Blackboard*) y las principales clases y relaciones que lo integran.



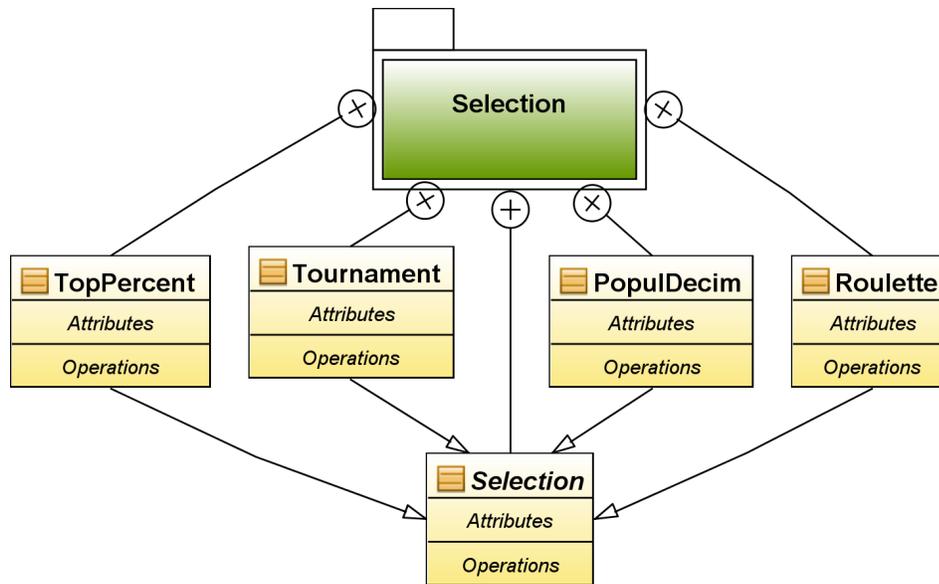
**Figura 7.59** El módulo *Conformation* y las principales clases y relaciones que lo integran.



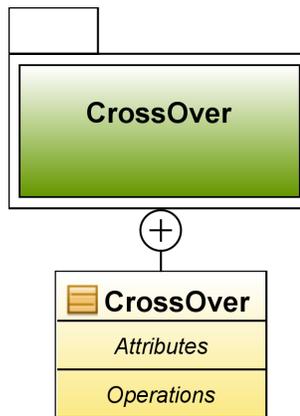
**Figura 7.60** El módulo *Generation* y las principales clases y relaciones que lo integran.



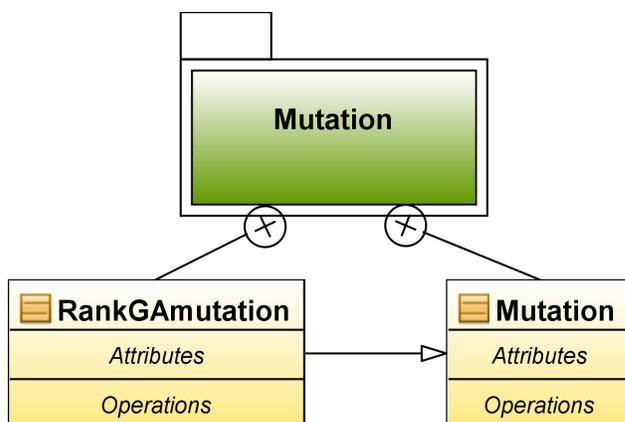
**Figura 7.61** El módulo *Algorithm* y la clase *Algorithm*.



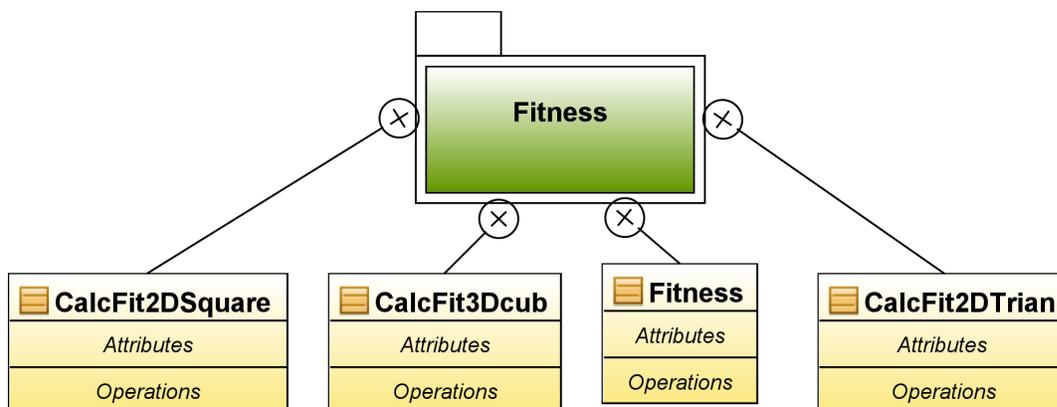
**Figura 7.62** El módulo *Selection* y las cuatro clases que lo integran. Cada clase representa un tipo de operador de Selección, invocado por el algoritmo de optimización (clase *Algorithm*).



**Figura 7.63** El módulo *Crossover* y la clase *Crossover*. Cada tipo de operador de crossover extiende de la clase *Crossover*.



**Figura 7.64** El módulo *Mutation* y las clases que lo integran. Cada clase que representa un diferente tipo de operador de mutación extiende de la clase *Mutation*.



**Figura 7.65** El módulo *Fitness* y las clases que lo integran. Cada una de las clases *CalcFit2DSquare*, *CalcFit2DTrian* y *CalcFit3Dcub* proporciona un método para calcular el fitness de la conformación (clase *Fitness*), en dependencia del tipo de espacio de plegamiento utilizado.

Las Figuras 7.66 a la 7.70 muestran aspectos del diseño detallado de las principales clases agrupadas en algunos de los módulos antes vistos.

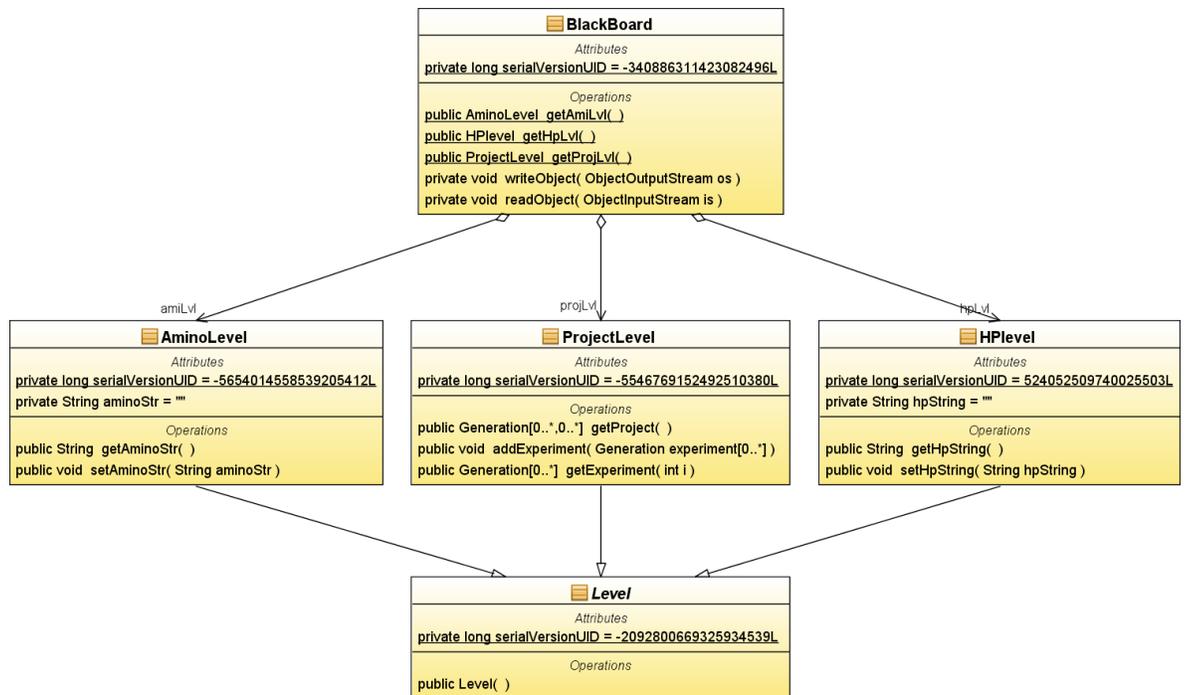
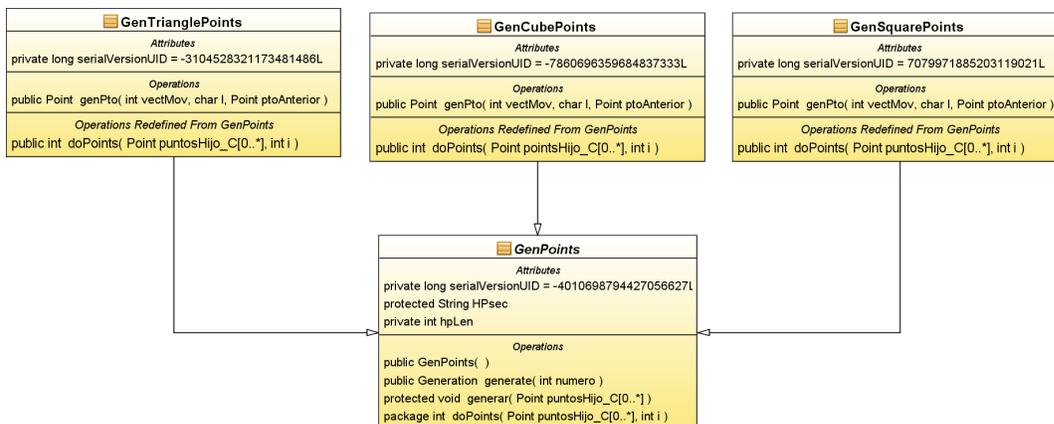


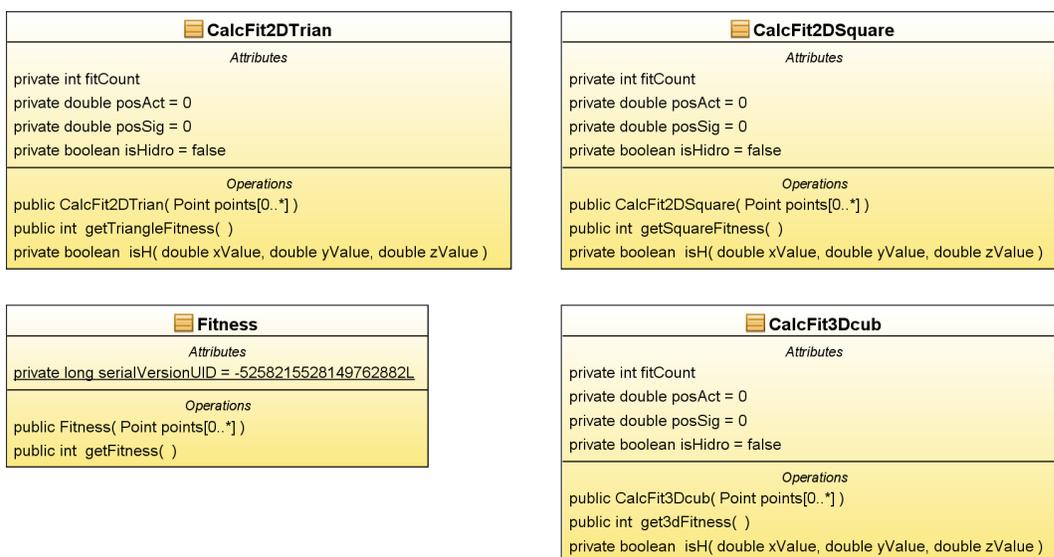
Figura 7.66 Diseño detallado de la clase *Blackboard* y de los niveles que la componen.



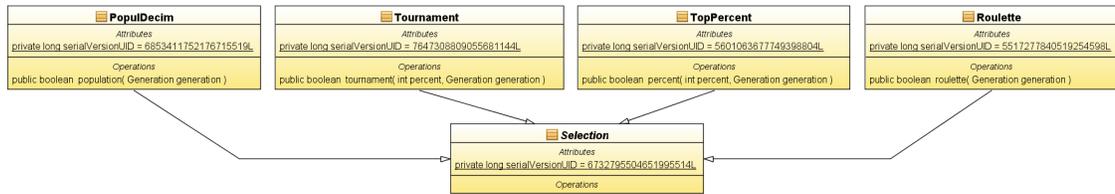
Figura 7.67 Diseño detallado de la clase *Conformation* como contenedor de las clases *Parents* y *Point*.



**Figura 7.68** Diseño detallado de las clases encargadas de generar los tres tipos de plegamiento de la secuencia HP objetivo, es decir, plegamiento en espacios 2D Cuadrado, 2D Triangular y 3D Cúbico.



**Figura 7.69** Diseño detallado de las clases *Fitness*, *CalcFit2DSquare*, *CalcFit2DTrian* y *CalcFit3Dcub*.



**Figura 7.70** Diseño detallado de las clases que representan los diferentes tipos de operadores de selección. Nótese que todos los operadores de selección extienden la clase *Selection*.

## VII.5 Referencias del capítulo

1. Cusumano, M.A. The Factory Approach to Large-Scale Software Development: Implications for Strategy, Technology, and Structure. Classic Reprints Series, 2017.
2. Garland, J., Anthony, R. Large Scale Software Architecture. A Practical Guide Using UML. Wiley, 2003.
3. Janakiram, D. Building Large Scale Software Systems. McGraw Hill Education, 2013.
4. Lakos, J. Large-Scale C++ Software Design. Addison Wesley, 1996.
5. Screerer, A. Coordination in Large-Scale Agile Software Development: Integrating Conditions and Configurations in Multiteam Systems (Progress in IS). Springer, 2017.
6. Fowler, M., Scott, K. UML Gota a Gota. Pearson Addison Wesley. 1999.
7. Rumbaugh, J., Jacobson, I., Booch, G. El Lenguaje Unificado de Modelado. Manual de Referencia. Pearson Addison Wesley. 2004.

# Capítulo VIII El uso de Patrones en la Ingeniería del Software: Patrones de Diseño

## VIII.1 El uso de patrones en la ingeniería del software

En Ingeniería de Software un patrón se refiere a la forma en la que un determinado problema de análisis, diseño, arquitectura, implementación, etc. fue solucionado, y cómo reutilizar la esencia de esta solución para resolver nuevos problemas. Un patrón encierra una experiencia, un conocimiento en la solución de problemas previos. Los patrones son el resultado de la experiencia práctica, siendo utilizados en el desarrollo de aplicaciones donde emergen problemas para los cuales tales resultados representan soluciones [1-4]. En el desarrollo de software a gran escala, el uso de los patrones es determinante para garantizar la claridad, manejabilidad, reusabilidad y extensibilidad requerida tanto en el diseño de la arquitectura del sistema como en el diseño de los componentes [5-9].

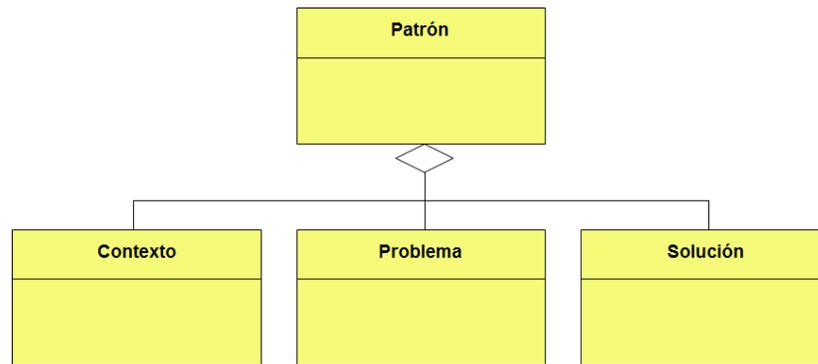
En Ingeniería de Software, un patrón:

- Maneja un problema de diseño (diseño arquitectónico o diseño de componentes) recurrente que aparece en situaciones de diseño específicas, presentando una solución a éste.
- Documenta experiencia de diseño existente que ha demostrado funcionar bien.
- Identifica y especifica abstracciones que están por encima del nivel de simples clases o instancias, o de componentes.
- Proporciona un vocabulario y comprensión comunes para principios de diseño.
- Es un medio para documentar arquitecturas de software.
- Ayuda a construir software complejo a gran escala.

- Ayuda a manejar la complejidad del software.

Como se ilustra en la Figura 8.1, en Ingeniería de Software un patrón es descrito como un esquema (*frame*) o descripción que consta de tres partes [2,3]:

- **Contexto.** La situación donde surge el problema.
- **Problema.** Descripción del problema que aparece repetidamente en el contexto dado.
- **Solución.** Muestra una solución ya probada para el problema.



**Figura 8.1** Descripción o componentes de un patrón en Ingeniería de Software.

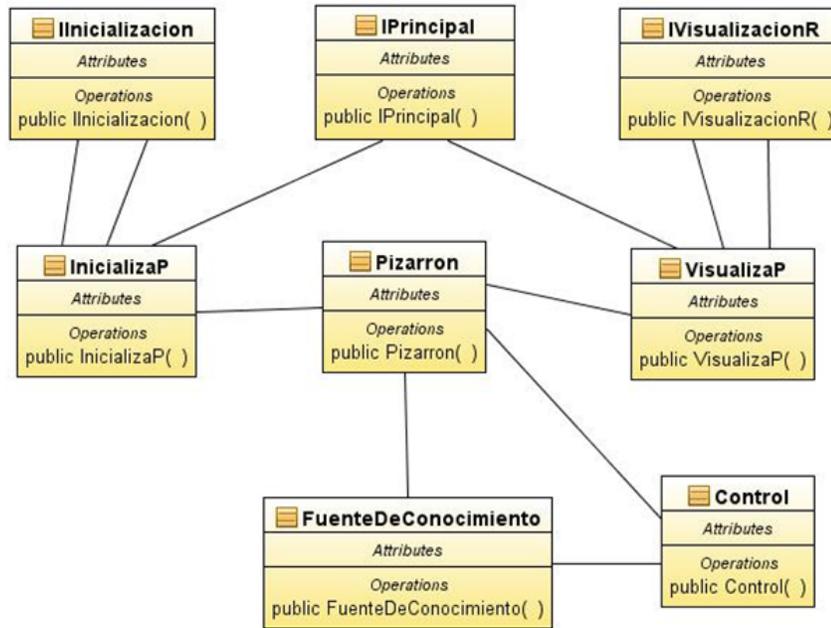
A continuación, veremos la descripción de dos patrones en dos dominios de problema. El primer ejemplo, ilustra la descripción de un patrón para la inicialización y visualización del pizarrón en una aplicación pizarrón (*blackboard architecture*). Por otra parte, el segundo ejemplo se refiere a la descripción de un patrón estructural para una red neuronal artificial.

### Patrón para la inicialización y visualización del pizarrón en una aplicación pizarrón (Blackboard Architecture)

**Contexto.** Aplicación basada en una arquitectura de pizarrón donde un grupo de fuentes de conocimiento se comunican a través de una estructura compartida para solucionar un problema. Cada fuente de conocimiento posee habilidad/conocimiento para solucionar una parte del problema. La solución del problema emerge como resultado del trabajo cooperativo de las fuentes de conocimiento. Un mecanismo de control decide el orden en el cual las fuentes de conocimiento ejecutan su acción sobre el pizarrón.

**Problema.** Determinados niveles del pizarrón requieren ser inicializados con información proporcionada por el usuario de la aplicación, a la vez que el usuario requiere la visualización de la información contenida en determinados niveles del pizarrón.

**Solución.** La solución de diseño se presenta en la Figura 8.2.



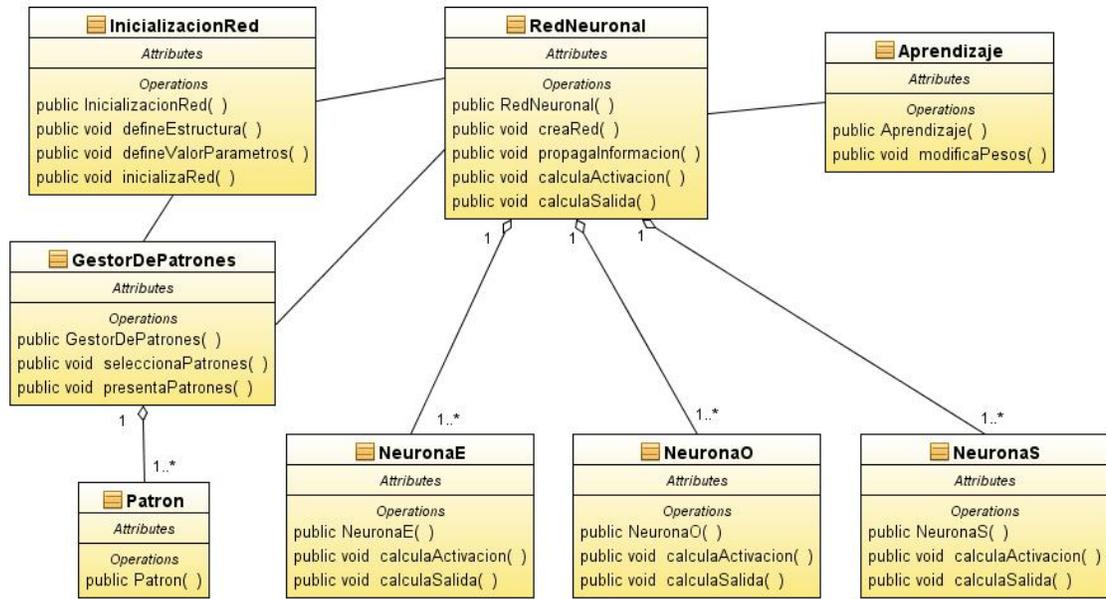
**Figura 8.2** Solución de diseño que propone el patrón para la inicialización y visualización del pizarrón en una aplicación blackboard.

### Patrón estructural de una red neuronal artificial

**Contexto:** Aplicaciones de software basadas en modelos neuronales para la solución de problemas de reconocimiento de patrones, extracción de patrones, aprendizaje, memoria, clasificación y agrupamiento de patrones.

**Problema:** Diseño de la estructura de un modelo de red neuronal considerando los componentes que lo integran –estratos de neuronas, neuronas, componente de inicialización de los parámetros de la red neuronal, componente para la presentación y propagación de los patrones de entrada, componente de aprendizaje, entre otros componentes. – y las relaciones entre estos componentes.

Solución. La solución de diseño se presenta en la Figura 8.3.



**Figura 8.3** Solución de diseño que propone el patrón estructural para una red neuronal artificial.

## VIII.2 Taxonomías de patrones

Como se puede apreciar en la Figura 8.4, en Ingeniería de Software se han identificado y propuesto patrones para solucionar problemas en las cuatro principales fases de desarrollo de sistemas de software: análisis, diseño arquitectónico, diseño detallado e implementación. Por el enfoque propio del presente material, aquí solo nos referiremos a los patrones de arquitectura y a los patrones de diseño. Nótese que los principales patrones de arquitectura fueron ya tratados en los capítulos V y VI, por lo que sólo nos enfocaremos en este capítulo en los patrones de diseño.

La Figura 8.5 muestra las categorías de patrones de arquitectura propuestas por Buschmann en [3]. Nótese que los patrones de la categoría “Sistemas Interactivos”, *Model-View-Controller* y *Presentation-Abstraction-Control*, así como el patrón de la categoría “Organización de la estructura”, *Blackboard*, fueron ya tratados en los capítulos V y VI. En la Figura 8.6 se ilustran las categorías de patrones de diseño según [3]. De esta clasificación de patrones, trataremos las categorías “Descomposición estructural”, “Organización del trabajo” y “Control de acceso”. Finalmente, la Figura 8.7 muestra la clasificación de patrones de diseño propuesta por Gamma en [2]. De esta clasificación nos referiremos a las categorías “Patrones de creación” y “Patrones estructurales”.

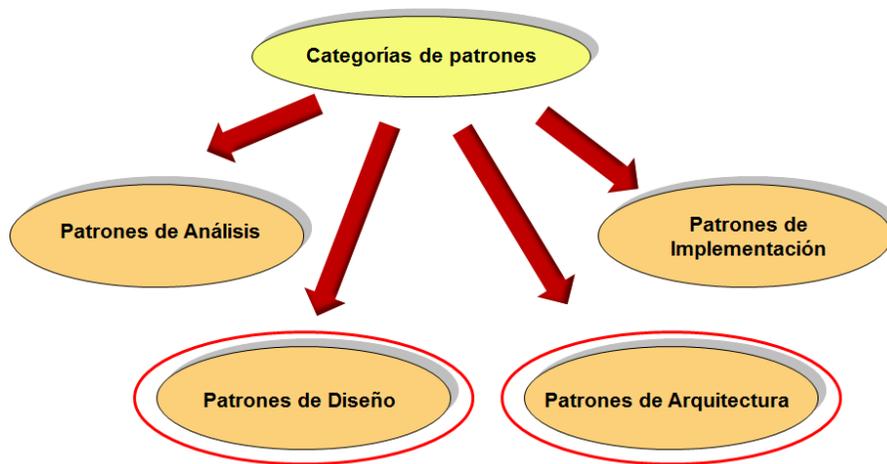


Figura 8.4 Taxonomía de patrones en Ingeniería de Software.

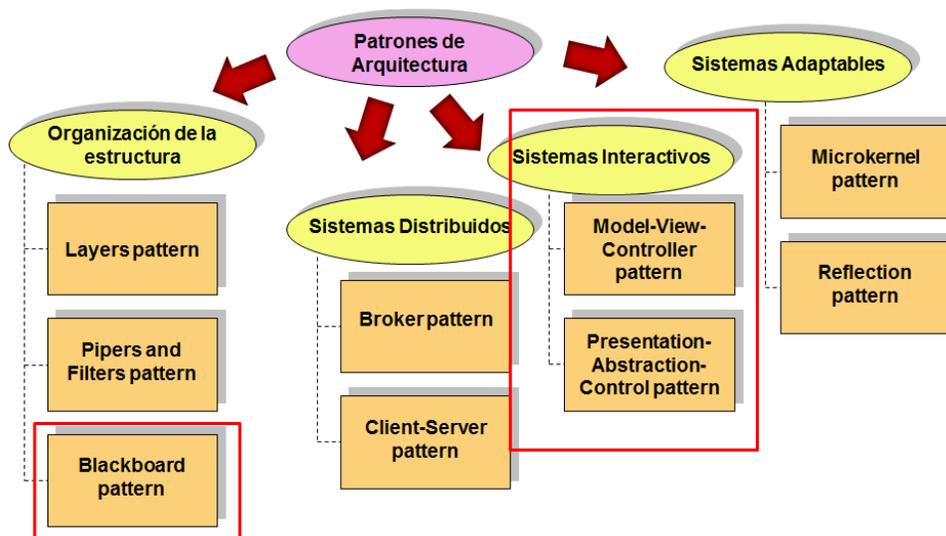


Figura 8.5 Categorías de patrones de arquitectura según Buschmann [3].

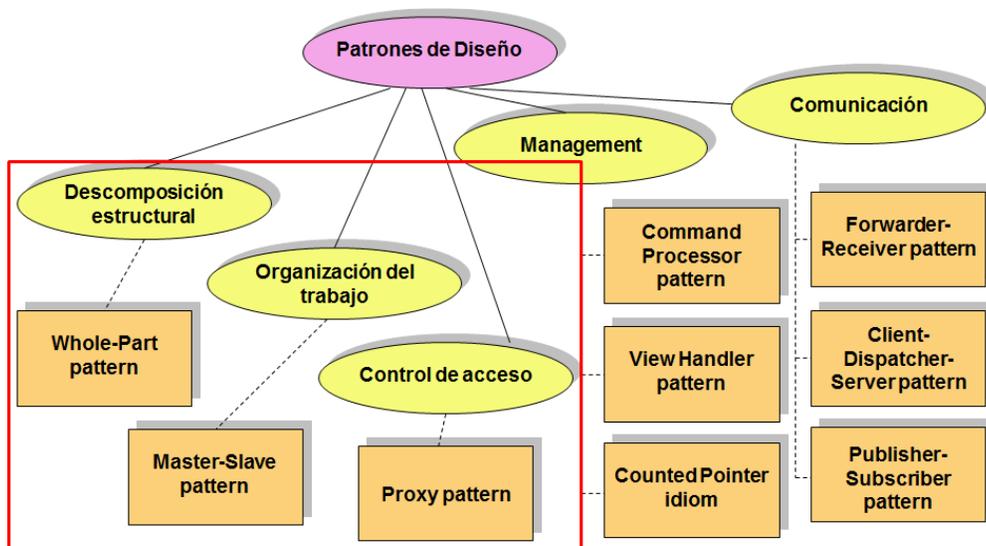


Figura 8.6 Categorías de patrones de diseño según Buschmann [3].

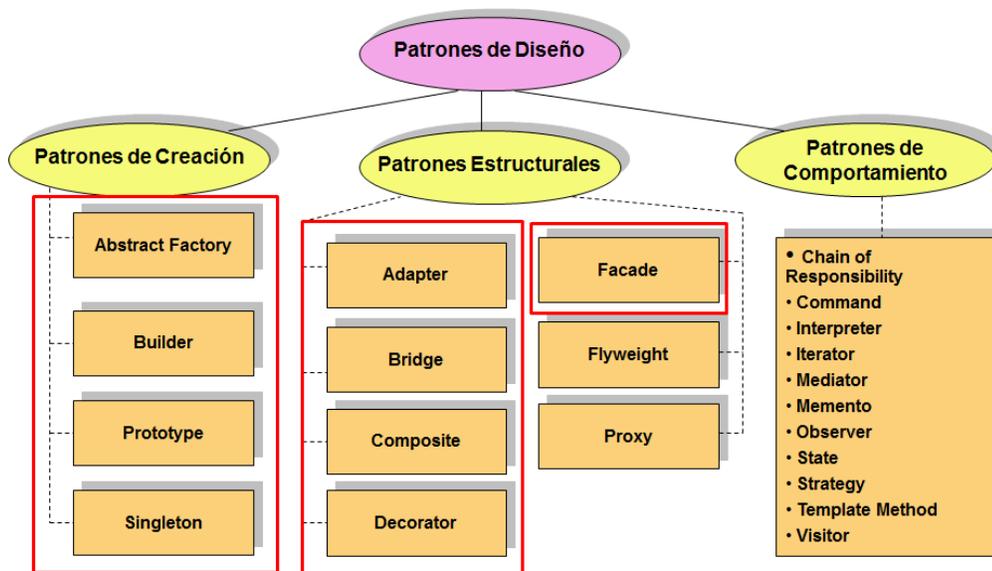


Figura 8.7 Categorías de patrones de diseño según Gamma [2].

### VIII.3 Patrones de diseño

Como su nombre lo indica, los patrones de diseño se refieren a la forma en la que un determinado problema de diseño de componentes o diseño detallado fue solucionado durante el desarrollo de un proyecto de software, y cómo reutilizar la esencia de esta solución para resolver nuevos problemas que se presenten durante la fase de diseño. Un patrón de diseño encierra una experiencia, un conocimiento en la solución de problemas relacionados con organización del trabajo, control de acceso, estructura, comportamiento, etc. En los siguientes epígrafes presentaremos y

discutiremos los patrones de diseño comúnmente usados en el desarrollo de software a gran escala, agrupándolos según las taxonomías ilustradas en las figuras 8.6 y 8.7.

### VIII.3.1 Patrones estructurales (Descomposición Estructural)

#### Patrón adaptador (Adapter pattern)

La utilidad del patrón Adaptador [2] consiste en que convierte la interfaz de una clase en otra interfaz que es la que esperan los clientes. De esta forma, permite que cooperen clases que no podrían cooperar por tener interfaces incompatibles. En otras palabras: un adaptador proporciona una interfaz estable para objetos parecidos con diferentes interfaces, convirtiendo la interfaz original de un componente en otra interfaz a través de un objeto adaptador intermedio. El patrón Adaptador comúnmente es usado cuando:

- Se quiere usar una clase existente y su interfaz no concuerda con la que se necesita.
- Se quiere crear una clase reutilizable que coopere con clases no relacionadas o que no han sido previstas, es decir, clases que no tienen interfaces compatibles.
- No se quiere, por ningún motivo, modificar el código existente.

El patrón adaptador puede ser utilizado como adaptador de clases o adaptador de objetos. Como se ilustra en la Figura 8.8, el adaptador de clases usa la herencia y la implementación de interfaces para adaptar una interfaz a otra. Por otra parte, como se puede apreciar en la Figura 8.9, el adaptador de objetos se basa en la composición de objetos.

Como se puede apreciar en las Figuras 8.8. y 8.9, en el patrón Adaptador la colaboración se establece según la siguiente secuencia:

1. El cliente invoca operaciones de una instancia de Adaptador.
2. El Adaptador invoca operaciones de Adaptable, que son realmente las que satisfacen la operación.

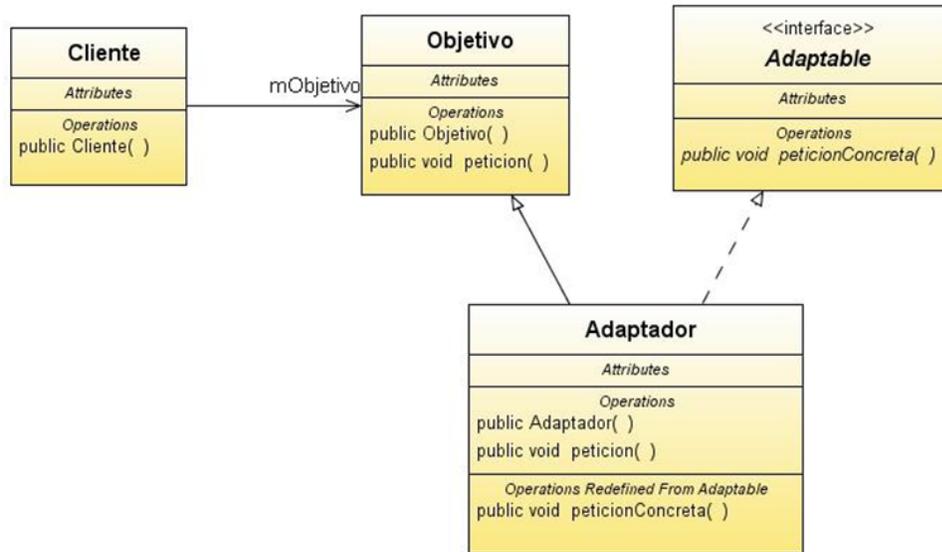


Figura 8.8 Patrón Adaptador basado en la adaptación de clases.

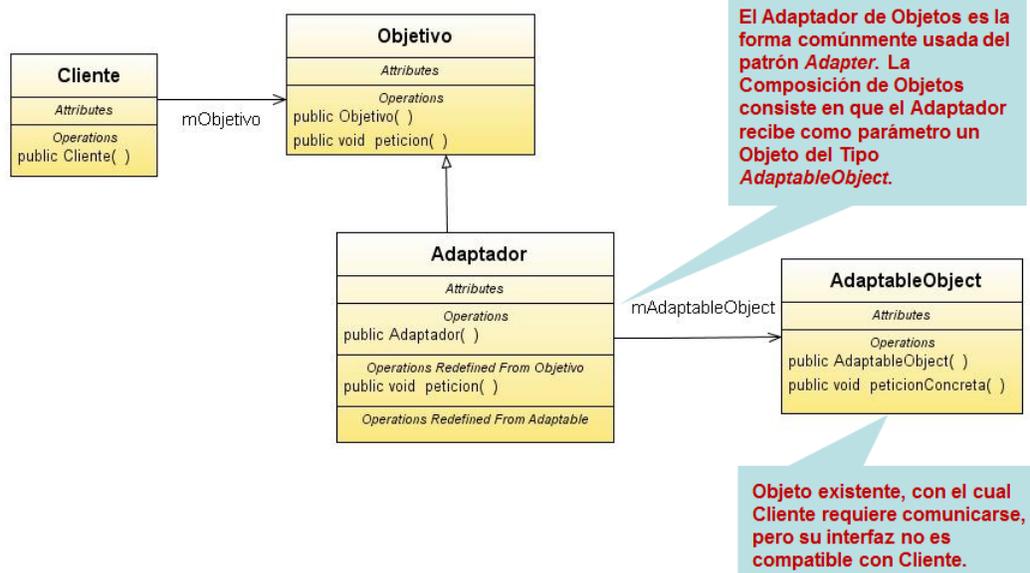


Figura 8.9 Patrón Adaptador basado en la adaptación de objetos.

Las Figuras 8.10 y 8.11 muestra el uso del patrón Adaptador en un fragmento de diseño correspondiente a un sistema de pagos. En ambos casos, el propósito fue que la clase *Cliente* fuera capaz de efectuar otros tipos de pago, adaptando la interfaz de una clase requerida pero no compatible con la clase *Cliente*, a una interfaz con la cual dicha clase pueda interactuar. En la Figura 8.10, la variante utilizada fue la adaptación de objetos. En este concepto de diseño la nueva clase *PagoMultiple* funge como el adaptador de objetos y recibe como parámetro un objeto del tipo *PagoTiempoAire*. Por otra parte, en la Figura 8.11, la variante seguida fue la

adaptación de clases. En este concepto de diseño la nueva clase *PagoMultiple* funge como el adaptador de clases, extendiendo la clase *Pago* e implementando la interfaz de la clase *PagoTiempoAire*. De esta forma, la clase *PagoTiempoAire* resulta ahora compatible con la clase *Cliente*.

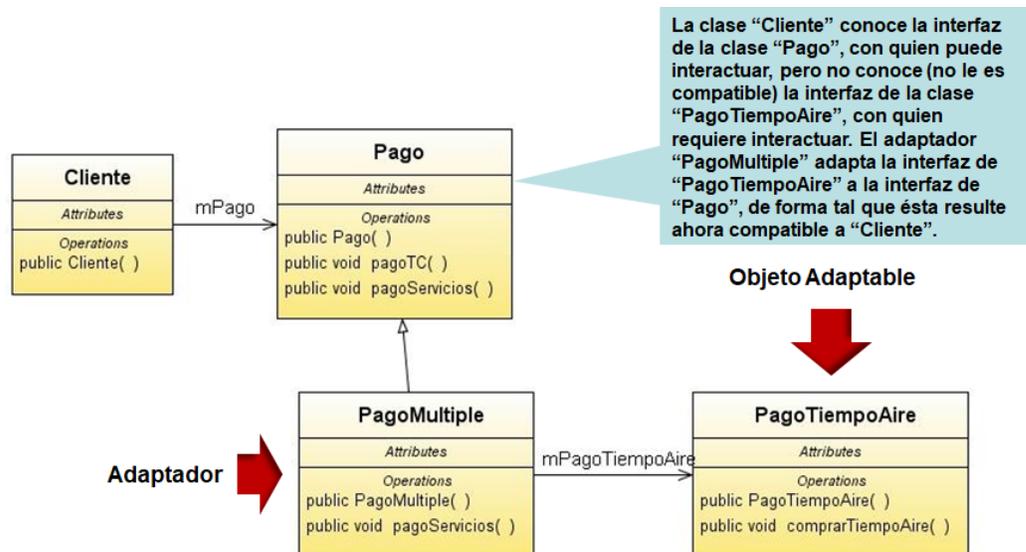


Figura 8.10 Ilustración del uso del patrón Adaptador, basado en la adaptación de objetos, en el diseño de un sistema de pagos.

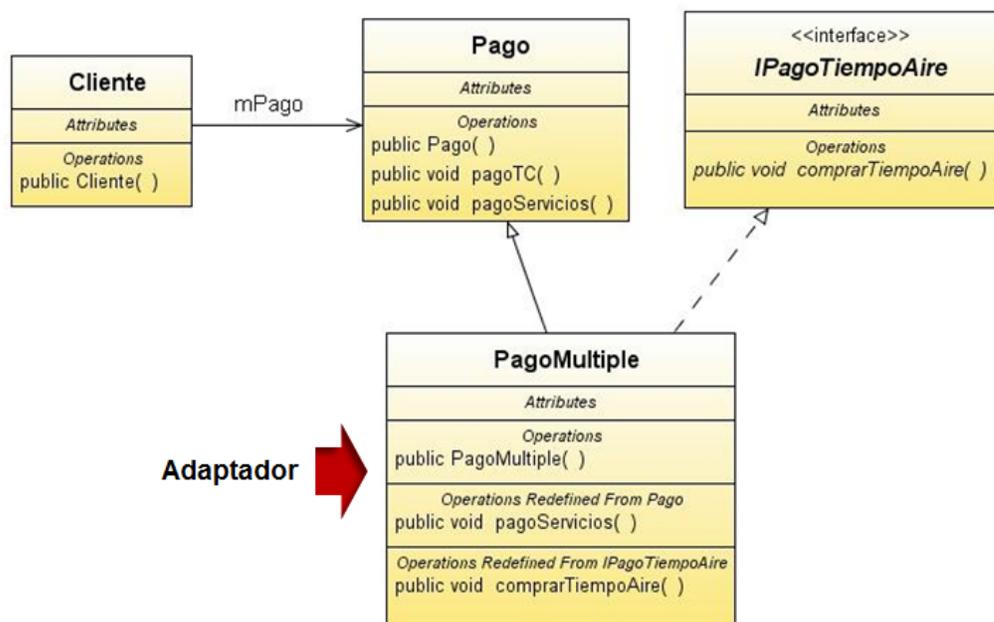
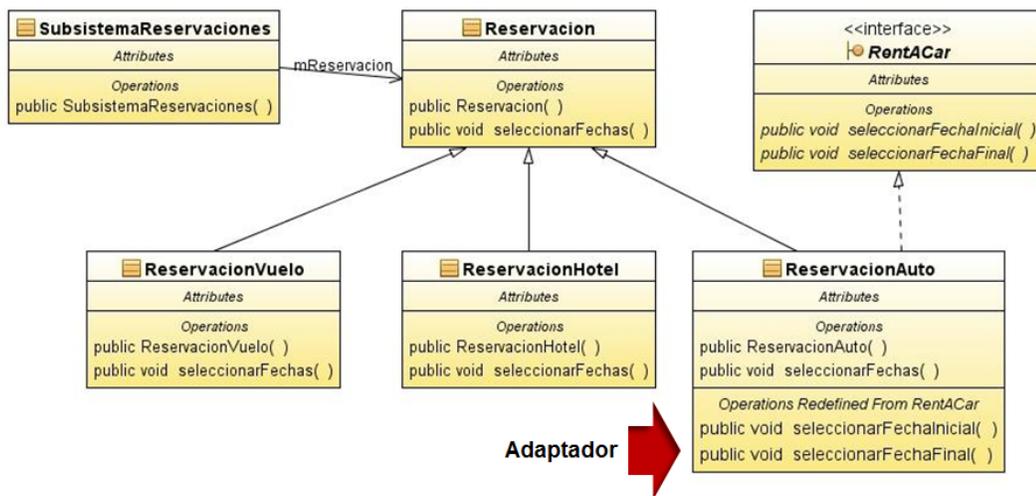
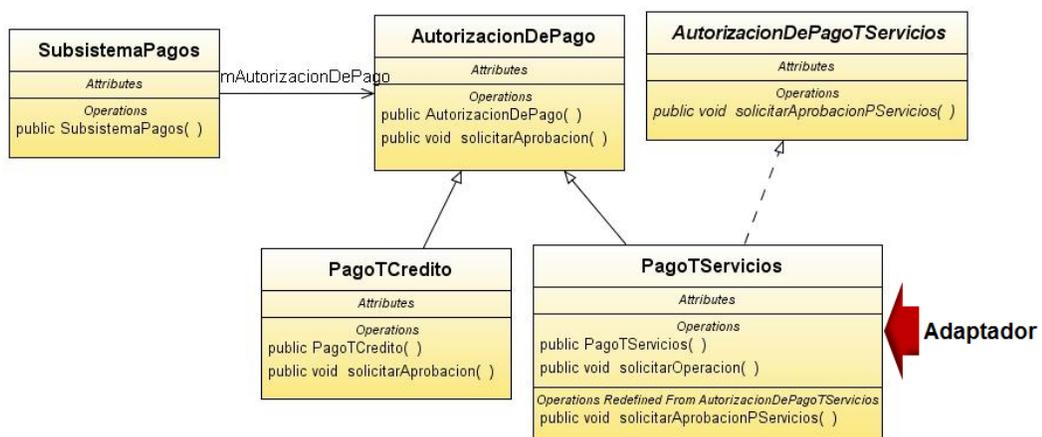


Figura 8.11 Ilustración del uso del patrón Adaptador, basado en la adaptación de clases, en el diseño del sistema de pagos ilustrado en la Figura 8.10.

La Figura 8.12 muestra el uso del patrón Adaptador en un fragmento de diseño correspondiente al sistema de reservaciones de una aerolínea. En este caso, el propósito fue que la clase *SubsistemaReservaciones* fuera capaz de efectuar, además de las reservaciones de hotel y vuelo, la reservación de auto, lo cual se logró a través de la adaptación de clases. Nótese que la clase *ReservacionAuto* funge como adaptador. Finalmente, en la Figura 8.13 se puede apreciar el uso del patrón Adaptador en un fragmento de diseño correspondiente a un sistema de autorización de pagos. De esta forma, la clase *SubsistemaPagos* es capaz de efectuar, además del pago en base a tarjetas de crédito, el pago utilizando tarjeta de servicios. Nótese que la clase *PagoTServicios* funge como adaptador.



**Figura 8.12** Ilustración del uso del patrón Adaptador, basado en la adaptación de objetos, en el diseño de un sistema de reservaciones de una aerolínea.



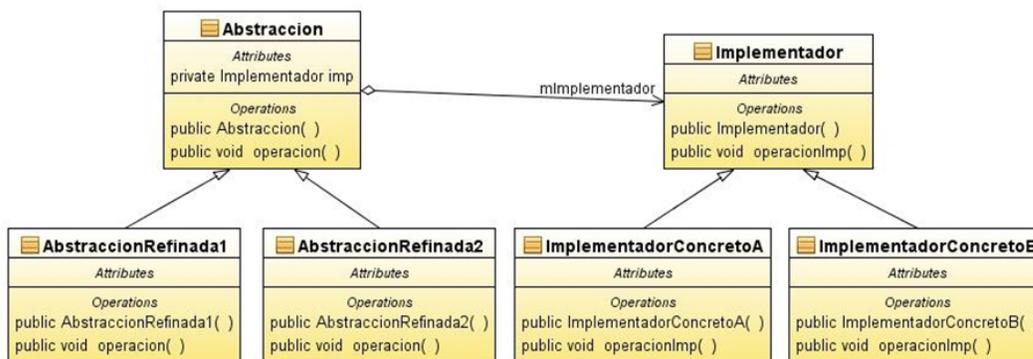
**Figura 8.13** Ilustración del uso del patrón Adaptador, basado en la adaptación de clases, en el diseño de un sistema de reservaciones de una aerolínea.

## Patrón Bridge

El patrón *Bridge* [2] es usado para desacoplar una abstracción de su implementación, de forma tal que ambas puedan variar de forma independiente. El patrón Bridge es comúnmente usado cuando:

- Se desea evitar un enlace permanente entre una abstracción y su implementación. Por ejemplo, cuando es necesario cambiar la implementación en tiempo de ejecución.
- Se requiere que tanto las abstracciones como sus implementaciones se puedan extender mediante subclasses. Las diferentes abstracciones pueden combinarse con las implementaciones, y ambas pueden extenderse de forma independiente.
- No se quiere, por ningún motivo, modificar el código existente.

Como se puede apreciar en la Figura 8.14, el patrón *Bridge* evita ligar una abstracción a una implementación a través de la creación de dos jerarquías de clases diferentes: una para las abstracciones y otra para las implementaciones.



**Figura 8.14** Patrón Bridge. Desacopla una abstracción de su implementación.

Las Figuras 8.15 y 8.16 ilustran el uso del patrón *Bridge* en dos situaciones donde era imprescindible desacoplar la abstracción de su implementación, de forma tal que diferentes implementaciones pudieran ser enlazadas en tiempo de ejecución a la abstracción. Como se ilustra en la Figura 8.15, en este caso el patrón *Bridge* es utilizado para desacoplar la abstracción *ReservacionHospedaje* de las tres formas diferentes en que se podría implementar, esto es Hotel, ApartHotel y Villa. De esta forma, en dependencia del tipo de reservación solicitada por la clase Cliente (la cual no se muestra en la Figura 8.15), en tiempo de ejecución se establecerá un enlace entre la abstracción *ReservacionHospedaje* y la implementación requerida. En la Figura 8.16 el patrón *Bridge* es utilizado en el diseño de un sistema experto de diagnóstico médico, para desacoplar la abstracción *Diagnóstico* de la forma en que este debe ser ejecutado en un momento dado, es decir, como diagnóstico genérico (*DGenerico*), etiológico (*DEtiologico*), sintomático (*DSintomatico*), etc., en dependencia del conocimiento médico experto que se requiera.

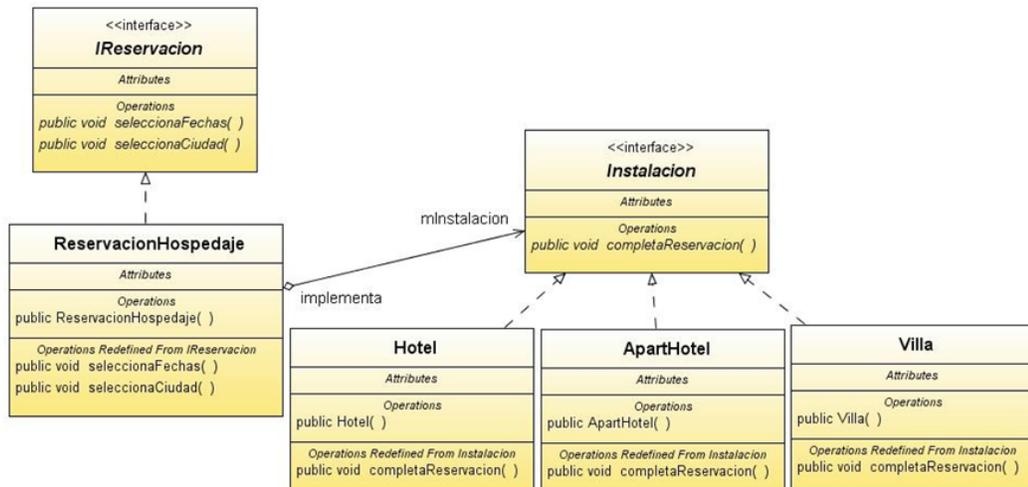


Figura 8.15 Uso del patrón Bridge en el diseño de un sistema de reservaciones de hospedaje.

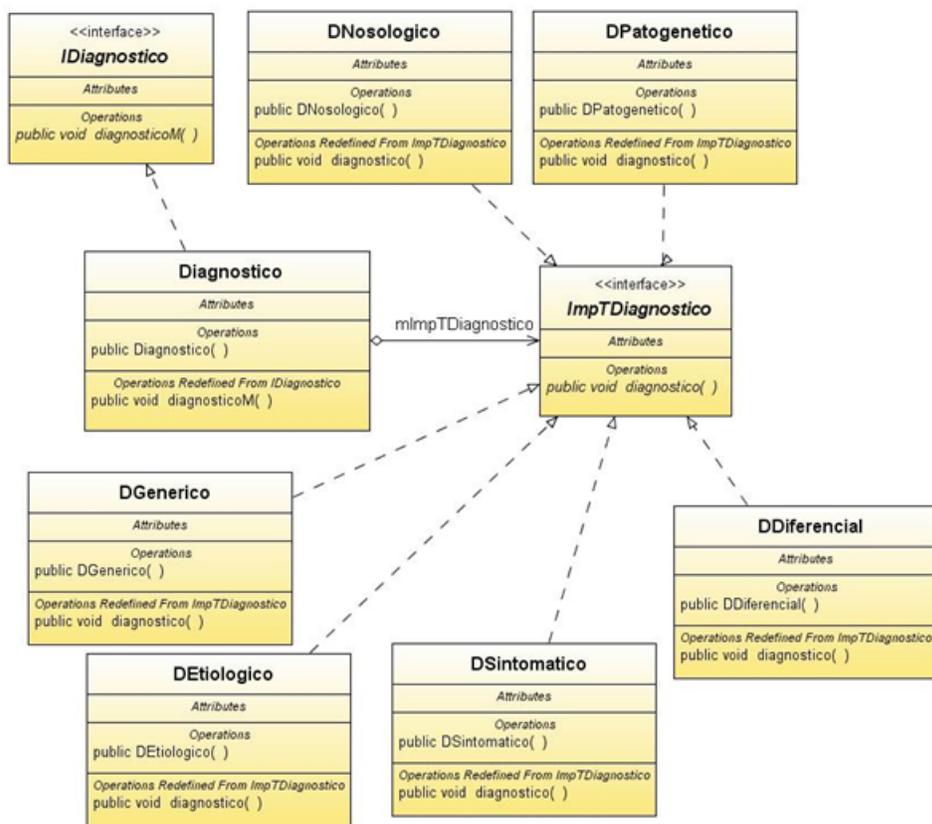


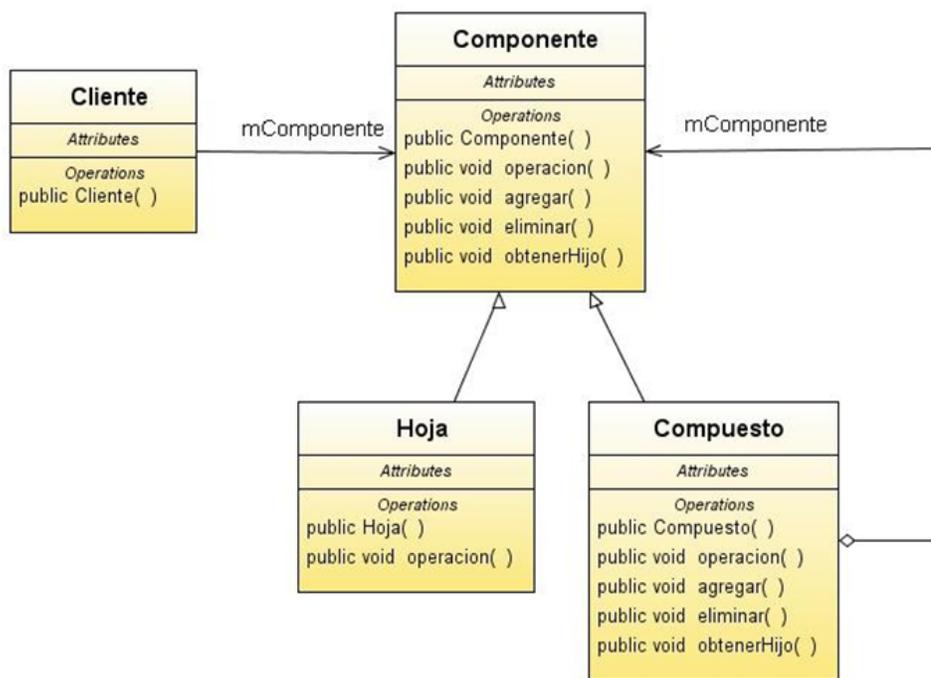
Figura 8.16 Uso del patrón Bridge en el diseño de un sistema experto de diagnóstico médico.

## Patrón Composite

El patrón *Composite* [2] proporciona una composición de objetos en estructura de árbol para representar jerarquías todo-parte, permitiendo que tanto los objetos simples como los compuestos sean tratados de manera uniforme. El patrón *Composite* es comúnmente utilizado cuando:

- Se necesita representar jerarquías de objetos todo-parte, donde un objeto parte puede ser a la vez ser una agregación de otras partes.
- Se desea tratar a todos los objetos de la estructura compuesta de manera uniforme.

Como se puede apreciar en la Figura 8.17, el patrón *Composite* define jerarquías de clases formadas por objetos simples y compuestos. Los objetos simples pueden componerse por otros objetos más complejos, los cuales a su vez pueden ser compuestos.



**Figura 8.17** Patrón *Composite* para la composición de objetos en estructura de árbol.

Las Figuras 8.18 y 8.19 ilustran dos ejemplos de uso del patrón *Composite*. Como se puede apreciar en la Figura 8.18, la jerarquía de clases fue utilizada para definir gráficos simples y gráficos compuestos. En este ejemplo, un gráfico compuesto podría contener uno o más rectángulos y una o más elipses. Por su parte, la Figura 8.19 ilustra el uso del patrón *Composite* en el diseño del sistema de reservaciones

de hospedaje ya visto. En este caso, una reservación compuesta (*ReservacionCompuesta*) es un agregado de diferentes tipos de reservaciones (*ReservacionVuelo*, *ReservacionHotel*, *ReservacionAuto*). El objeto *ReservacionCompuesta* como contenedor permite agregar/eliminar objetos hijos.

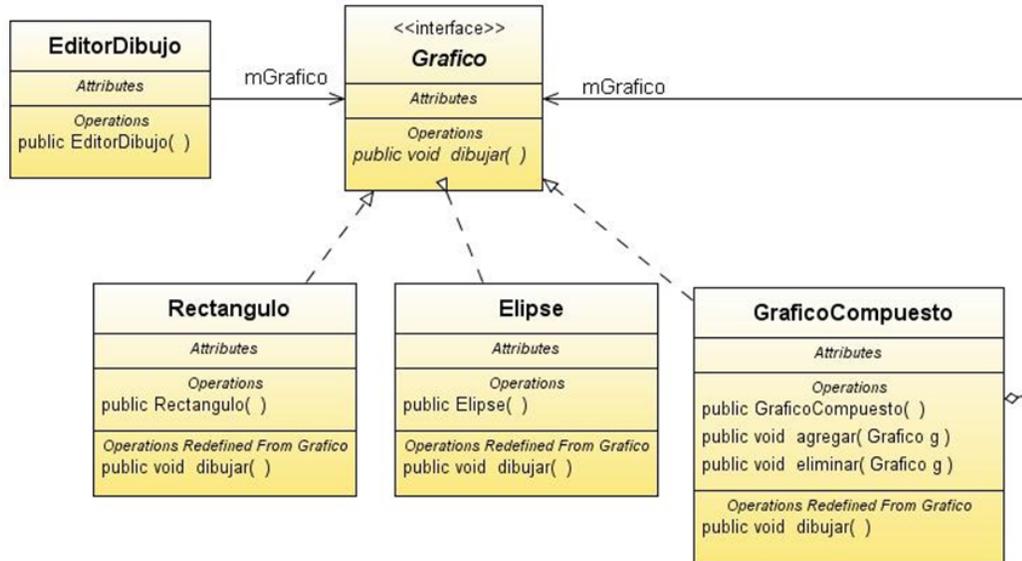


Figura 8.18 Uso del patrón *Composite* en el diseño de un sistema editor de figuras.

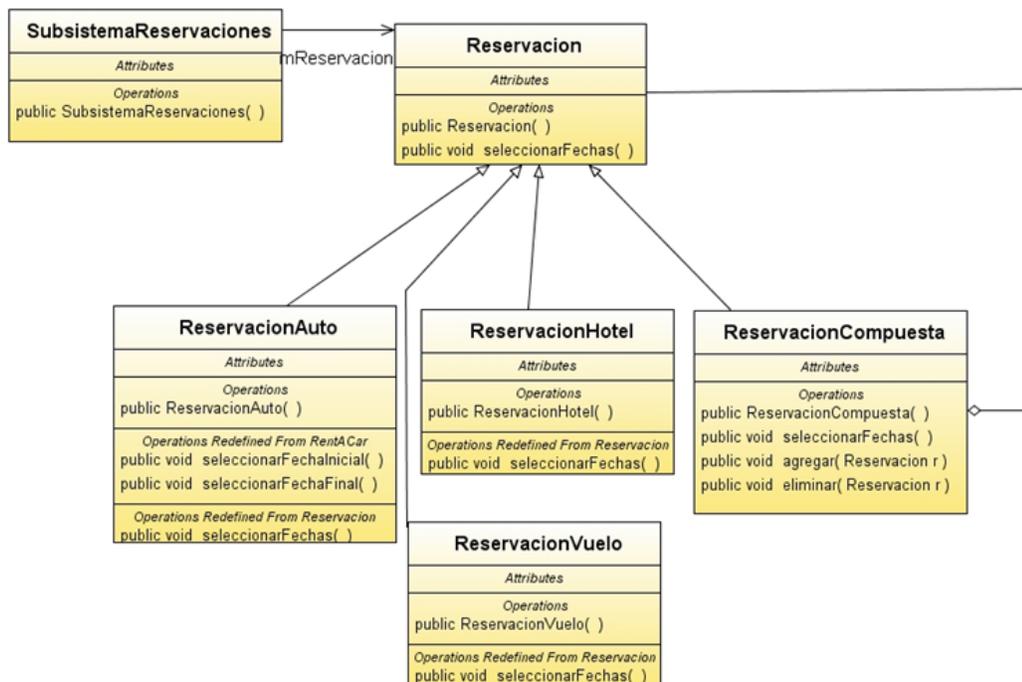
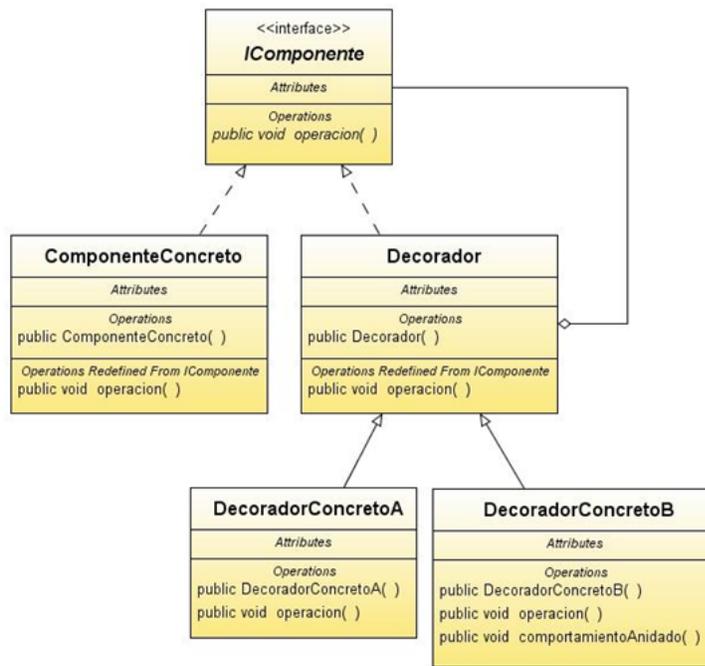


Figura 8.19 Uso del patrón *Composite* en el diseño de un sistema de reservaciones de hospedaje.

## Patrón Decorator

El patrón *Decorator* [2] permite asignar nuevos comportamientos a un objeto dinámicamente. De esta forma, el patrón proporciona una alternativa flexible a la herencia para extender la funcionalidad, “encerrando o envolviendo” el objeto original con el nuevo decorador. Como se puede apreciar en la Figura 8.20, lo anterior comúnmente se logra pasando el objeto original (*ComponenteConcreto*) como un parámetro al constructor del decorador (*Decorador*), y este último implementa la nueva funcionalidad. Comúnmente, el patrón *Decorator* es utilizado cuando:

- Se requiere añadir objetos individuales de forma dinámica y transparente, sin afectar a otros objetos existentes.
- La extensión a través de la herencia no es viable.



**Figura 8.20** El patrón *Decorator* como alternativa para añadir nuevos comportamientos a un objeto existente.

Las Figuras 8.21 y 8.22 ilustran el uso del patrón *Decorator*. Como se puede apreciar en la Figura 8.21, el patrón *Decorator* fue utilizado para añadir nuevo comportamiento al objeto *VentanaSimple*, el cual es enviado como parámetro al decorador *DecoradorVentana*, el cual le añade nuevas funcionalidades relacionadas con la incorporación a la ventana del *scrollbar* vertical y/o del *scrollbar* horizontal. Por otra parte, en la Figura 8.22 el patrón *Decorator* es utilizado en el diseño del ya conocido sistema de reservaciones de una aerolínea. En este caso, el patrón *Decorator* envuelve el objeto simple *ReservacionVuelo*, para añadirle otras

funcionalidades tales como las relacionadas con la reservación de hotel y la reservación de auto.

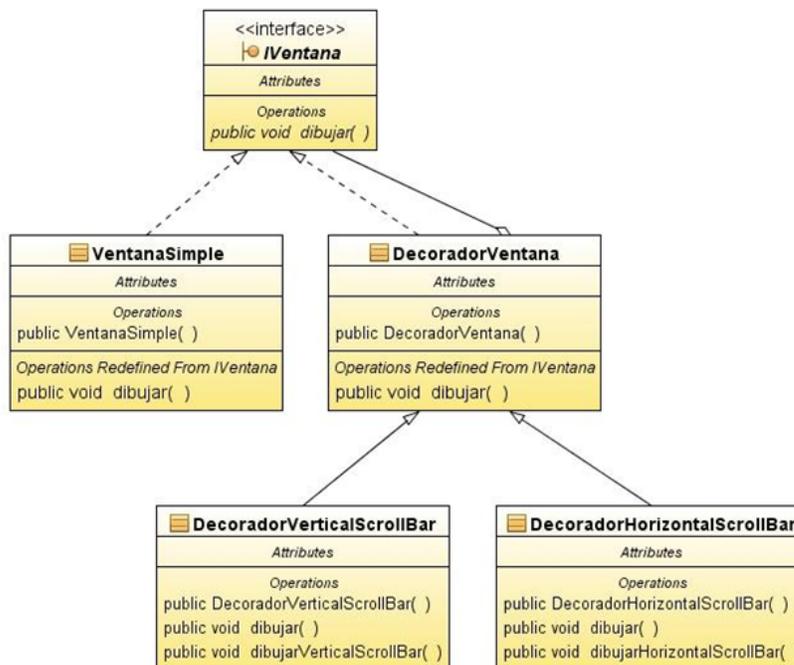


Figura 8.21 Uso del patrón *Decorator* en el diseño de las GUI de un sistema interactivo.

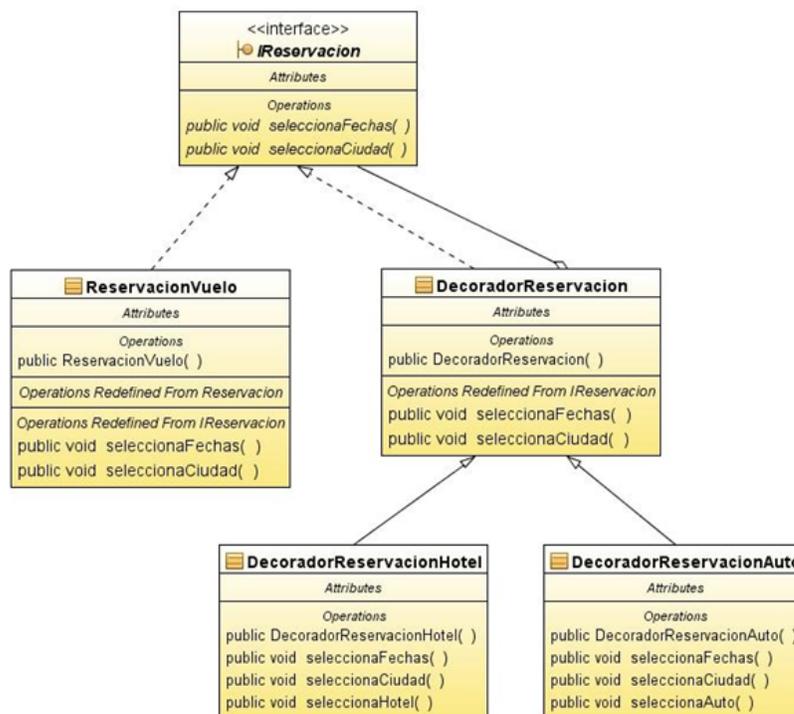
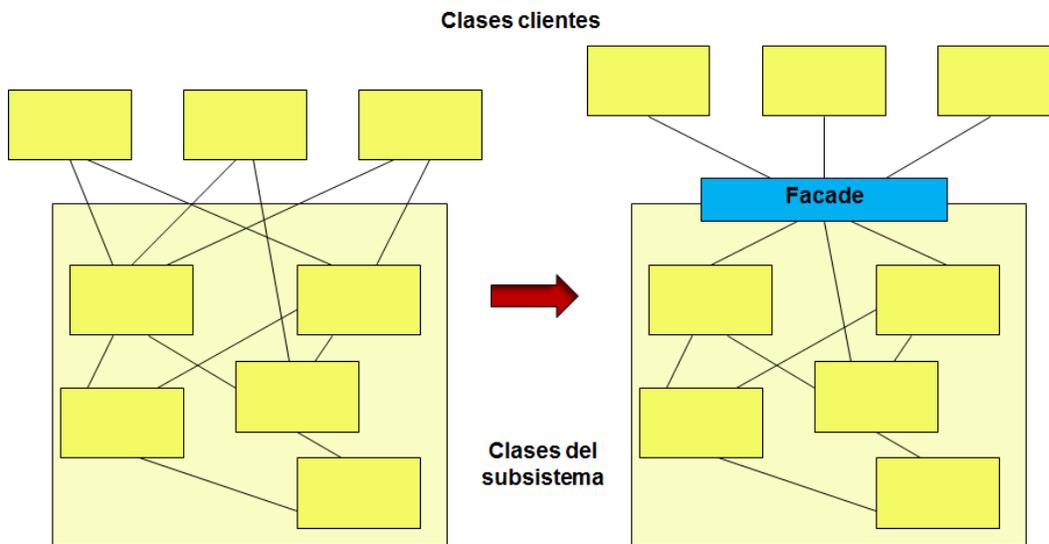


Figura 8.22 Uso del patrón *Decorator* en el diseño del sistema de reservaciones de una aerolínea.

## Patrón Facade

El patrón *Facade* proporciona una interfaz unificada para un conjunto de clases de un subsistema, definiendo una interfaz de alto nivel que hace que el subsistema sea más fácil de usar. Como se puede apreciar en las Figuras 8.23, 8.24 y 8.25, una *Facade* (fachada) es un objeto que proporciona una interfaz simplificada a un gran número de clases, tal como una biblioteca de clases. Como se puede apreciar en la Figura 8.25, el patrón *Facade* puede ser utilizado para proporcionar una interfaz unificada a un único subsistema compuesto a su vez por varios módulos. Comúnmente, el patrón *Facade* es utilizado cuando:

- Deseamos proporcionar una interfaz simple para un sistema complejo.
- Haya muchas dependencias entre las clases clientes y las clases que proporcionan servicios.
- Necesitamos dividir en capas nuestros subsistemas, usando una fachada para definir un punto de entrada en cada nivel del subsistema.



**Figura 8.23** El patrón *Facade* como objeto que proporciona una interfaz simplificada a las clases de un subsistema.

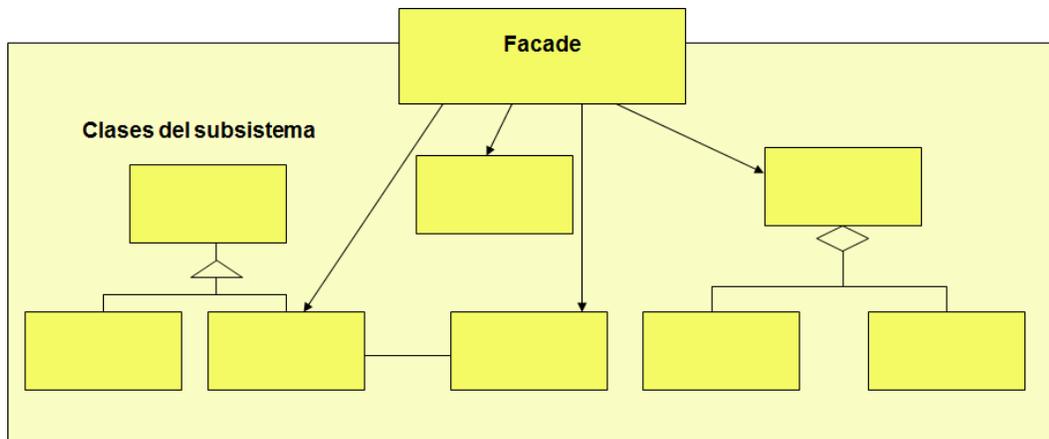


Figura 8.24 El patrón *Facade* (detalle).

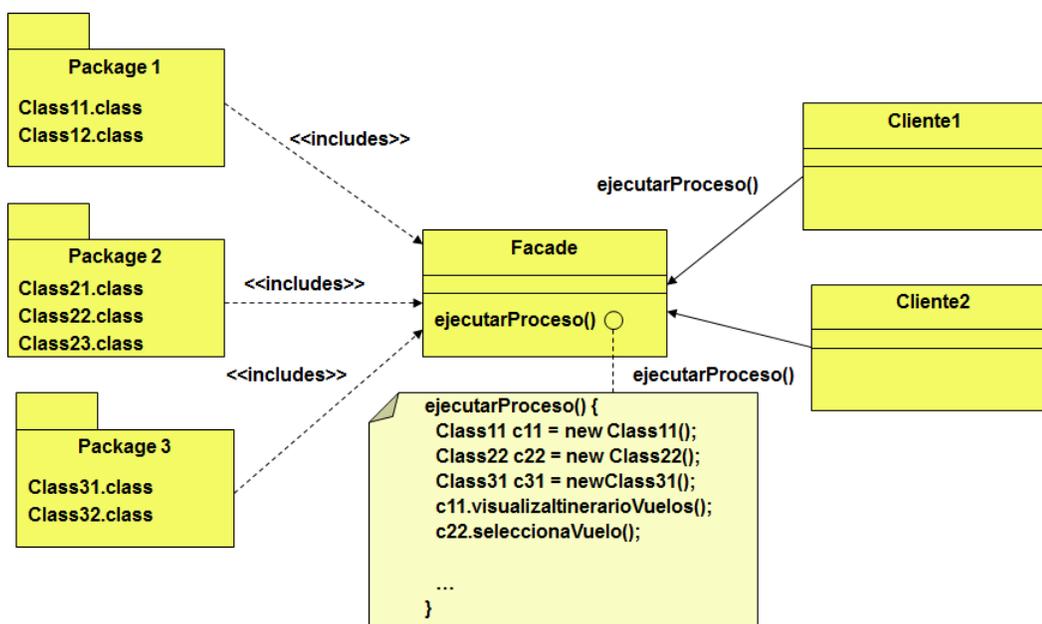
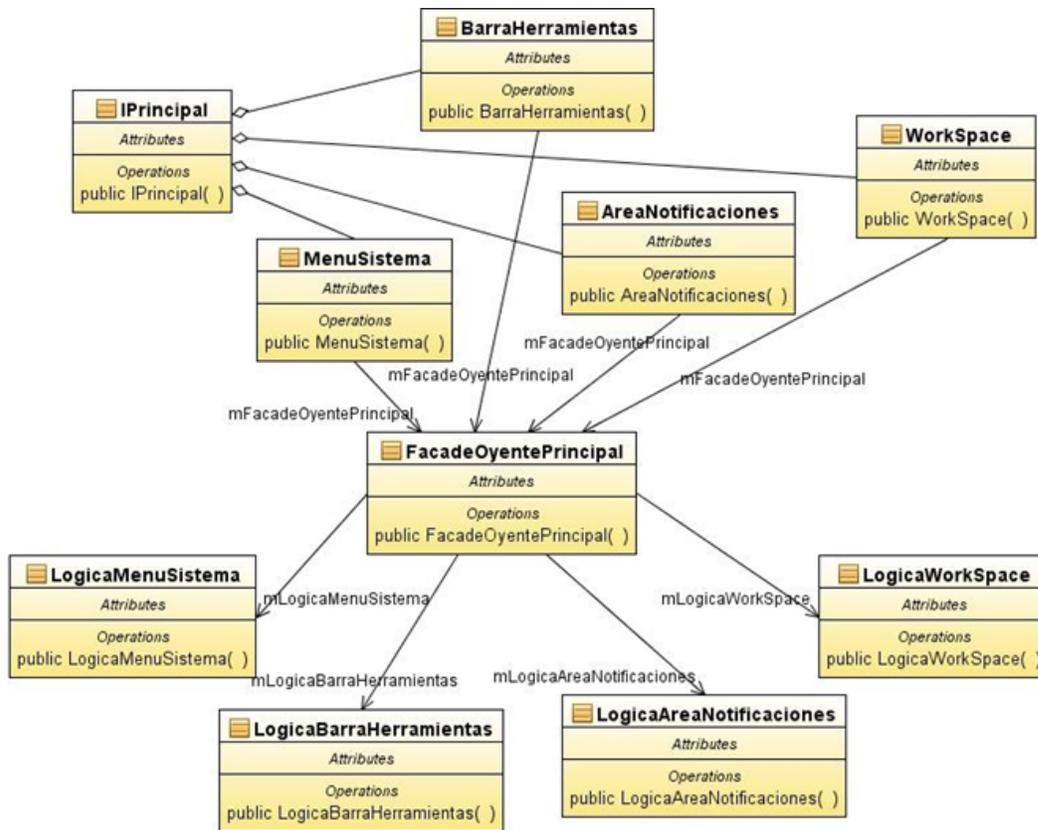


Figura 8.25 El patrón *Facade*. La fachada funciona como intermediario entre las clases Cliente y las clases que proporcionan servicios.

La Figura 8.26 ilustra el uso del patrón *Facade* en el diseño del sistema para la recuperación de proyectos de software, al cual nos hemos referido como ejemplo en algunos de los capítulos previos. Como se puede apreciar en esta figura, el patrón *Facade* (*FacadeOyentePrincipal*) proporciona una interfaz unificada que funge de intermediaria entre los componentes de la vista y los componentes del modelo (lógica).



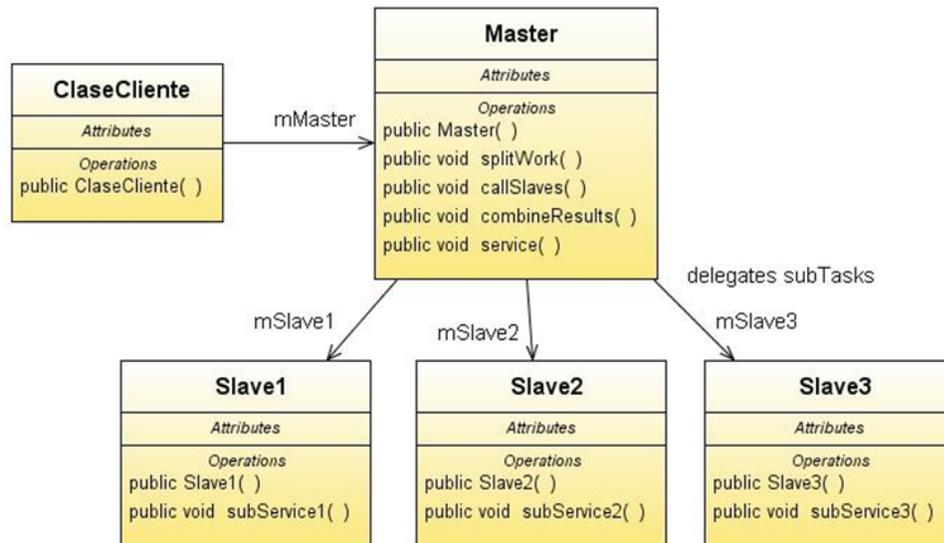
**Figura 8.26** Uso del patrón *Facade* en el diseño de un sistema para la recuperación proyectos de software. En este caso, el patrón *Facade* es usado como un superoyente en la arquitectura MVC, el cual proporciona una interfaz unificada a un subsistema de la lógica.

### VIII.3.2 Patrones de organización del trabajo

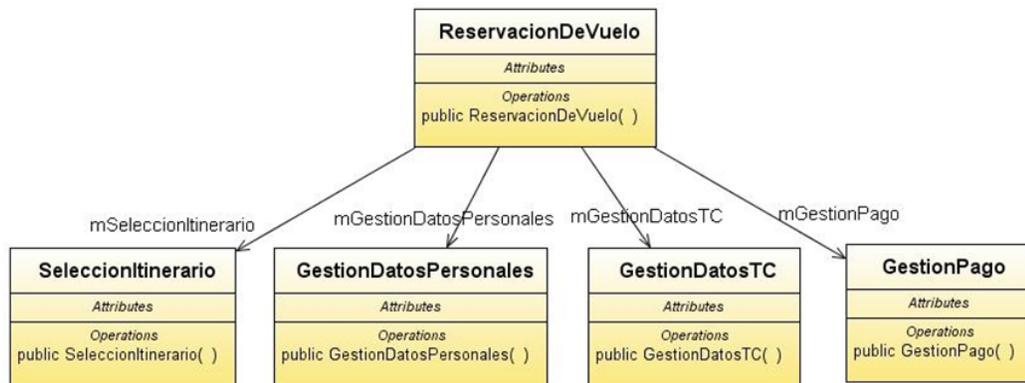
#### Patrón Master-Slave

El patrón *Master-Slave* es un patrón de organización del trabajo, que define un componente superordinado (*Master*) y un grupo de componentes subordinados (*Slave*). El componente *Master* distribuye el trabajo a los componentes *Slaves*, los cuales no tienen que ser necesariamente idénticos, y computa un resultado final a partir de los resultados retornados por los *Slaves*. El patrón *Master-Slave* ha sido comúnmente usado para servicios de computación concurrente y paralela, en este caso los *Slaves* podrían ser idénticos. La Figura 8.27 ilustra la estructura de clases que caracteriza al patrón *Master-Slave*. En la Figura 8.28 se puede apreciar un ejemplo del uso del patrón *Master-Slave* en el ya tratado sistemas de reservaciones de vuelos de una aerolínea. Como se puede apreciar en esta figura, la clase *ReservacionDeVuelo* funge como *Master*, delegando en sus *Slaves* tareas tales como selección del itinerario de vuelo, gestión de los datos personales, gestión de los datos de la tarjeta de crédito y gestión del pago. En este ejemplo, los *Slaves* ejecutan tareas diferentes, algunas de las cuales podrían ser concurrentes o secuenciales. Otros

ejemplos del uso del patrón *Master-Slave* son proporcionados al final de este capítulo, en el epígrafe dedicado a los casos de estudio SPI y Evolution.



**Figura 8.27** El patrón *Master-Slave* para la organización del trabajo.



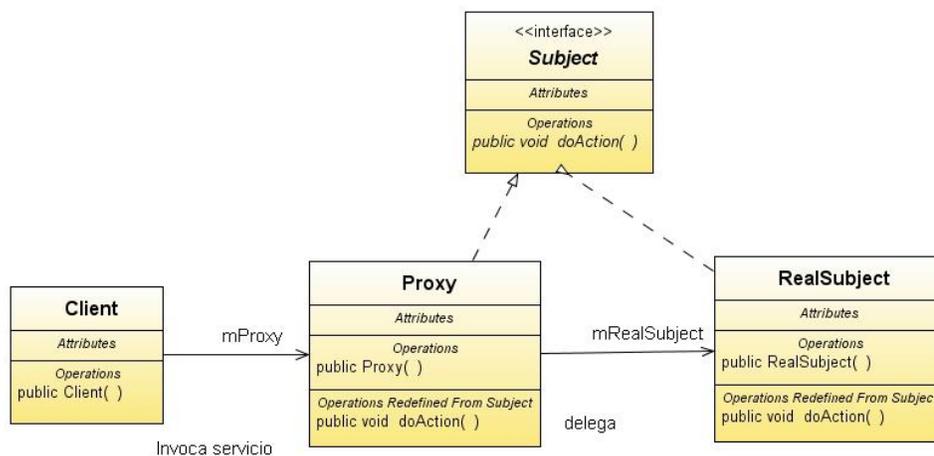
**Figura 8.28** Uso del patrón *Master-Slave* en el diseño de un sistema de reservaciones de vuelo de una aerolínea.

### VIII.3.3 Patrones de control de acceso

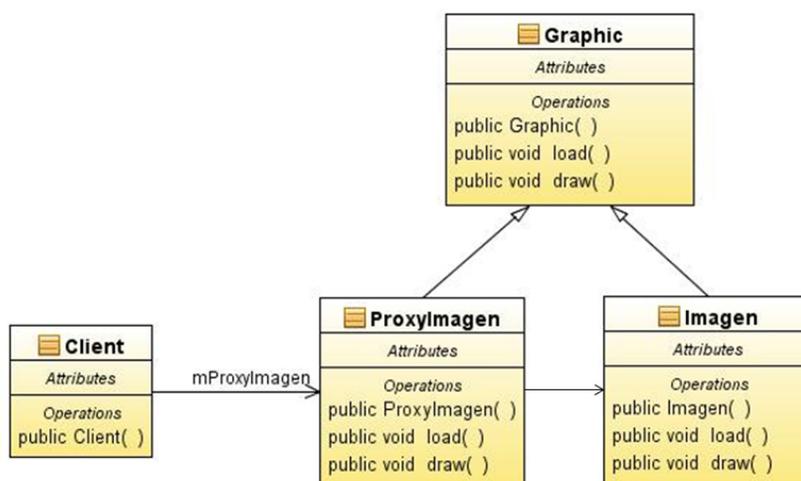
#### Patrón Proxy

El patrón *Proxy* es un patrón de control de acceso, donde el *Proxy* es comúnmente utilizado como un intermediario entre los componentes que requieren de un servicio y los componentes que proveen ese servicio. Este patrón ha sido utilizado tanto desde el punto de vista lógico como físico en el software y hardware, respectivamente. Como ya habíamos mencionado, el *Proxy* es una clase que funciona como una interfaz para acceder a los servicios proporcionados por otra

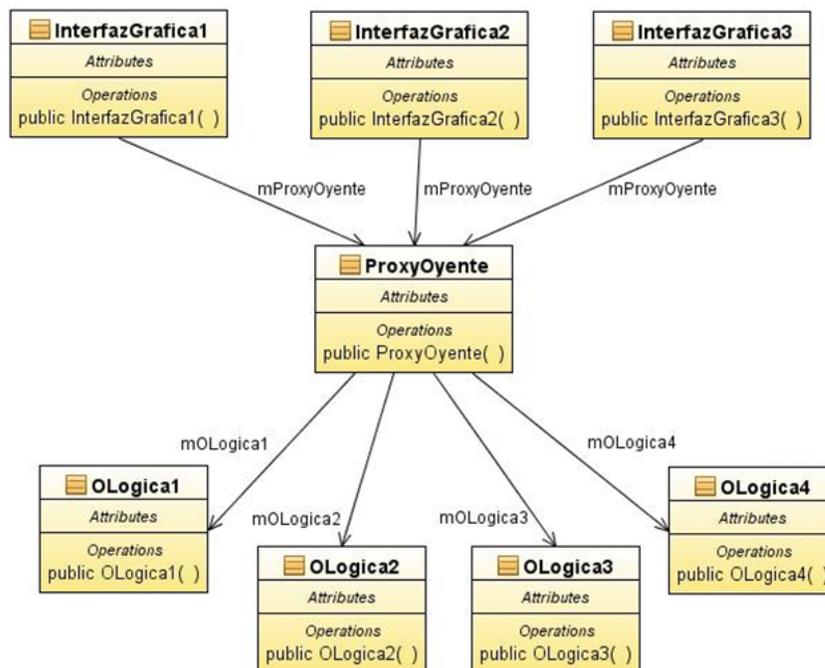
clase. En este sentido, podemos decir que la clase Cliente hace la solicitud de servicio al *Proxy*, quien delega las peticiones recibidas a la clase real que ofrece tales servicios. La Figura 8.29 muestra los componentes e interacciones que intervienen en el patrón *Proxy*. Por otra parte, la Figuras 8.30 y 8.31 ilustran dos ejemplos de aplicaciones del patrón *Proxy*. En la Figura 8.30 se muestra el uso del patrón *Proxy* en el componente de un sistema dedicado a la gestión de gráficos. Aquí la clase Cliente evita la interacción directa con el objeto Imagen, al solicitar sus requerimientos al objeto *Proxy* y éste delegarlos en el objeto Imagen. En la Figura 8.31, el patrón *Proxy* ha sido utilizado para simular una fachada (*Facade*) que unifica las interacciones hacia un conjunto de objetos de la lógica. El objeto *ProxyOyente* es notificado de los eventos que ocurren a nivel de un conjunto de GUI relacionadas de la capa Vista, y éste los traduce en peticiones hacia los objetos que proporcionan los servicios requeridos en la capa Modelo.



**Figura 8.29** El patrón *Proxy* como intermediario entre clases clientes y clases servidores.



**Figura 8.30** Uso del patrón *Proxy* en el diseño de un sistema que gestiona el almacenamiento y recuperación de imágenes.



**Figura 8.31** Uso del patrón *Proxy* como una fachada (*Facade*) en una arquitectura Modelo-Vista-Controlador.

### VIII.3.4 Patrones de creación

El denominador común de los patrones de diseño conocidos como patrones de creación, es que éstos permiten desacoplar la representación de la lógica (reglas de negocios del sistema) de la creación, composición y representación de objetos productos, comúnmente requeridos tanto en el comportamiento como en la lógica del sistema de software. Para lograr lo anterior, los patrones de creación proponen mecanismos que permiten:

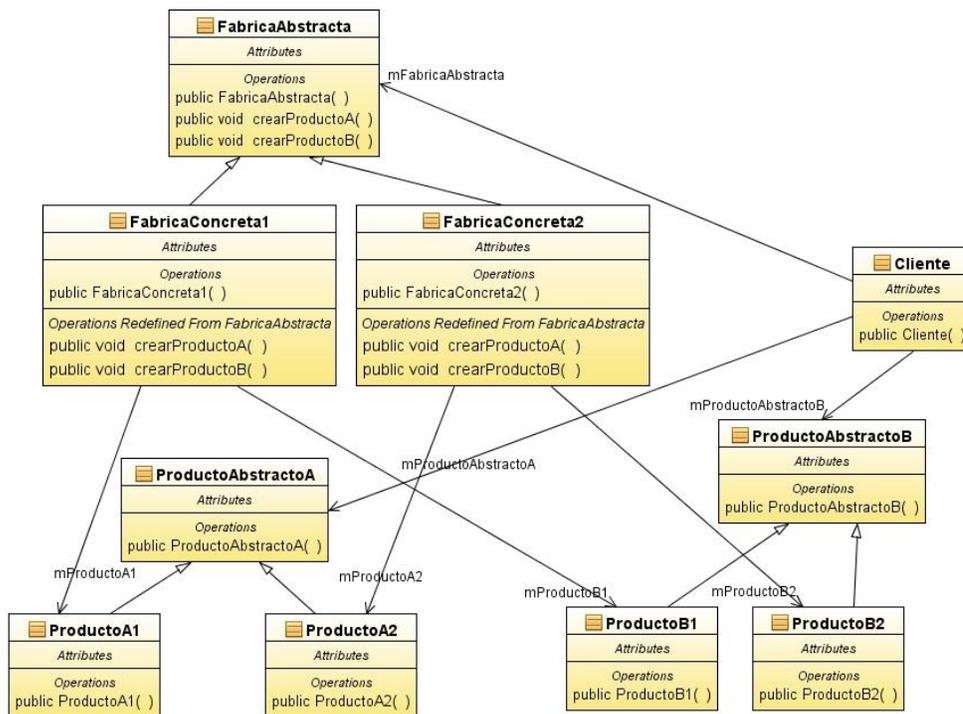
- Independizar la representación de la lógica de un sistema de la forma en que se construyen, componen y representan los objetos productos.
- Separar la construcción de un objeto completo de la forma en que éste es representado.
- Delegar en las subclasses la creación de objetos.
- Evitar la creación de nuevos objetos, utilizando para ello la copia o clonación de la instancia original.
- Imponer restricciones en el número de instancias permitidas de una determinada clase.

## Patrón Abstract Factory

El patrón de creación *Abstract Factory* [2] permite separar el comportamiento y la lógica de un sistema de la forma en que se construyen, componen y representan los objetos productos, al proporcionar una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas. Comúnmente, el patrón *Abstract Factory* es utilizado cuando:

- Un sistema debe ser independiente de cómo se crean, componen y representan sus productos (objetos).
- Un sistema debe ser configurado con una de entre varias familias de productos.
- Una familia de productos relacionados está diseñada para ser usada conjuntamente.
- Se desea proporcionar una biblioteca de clases de productos.

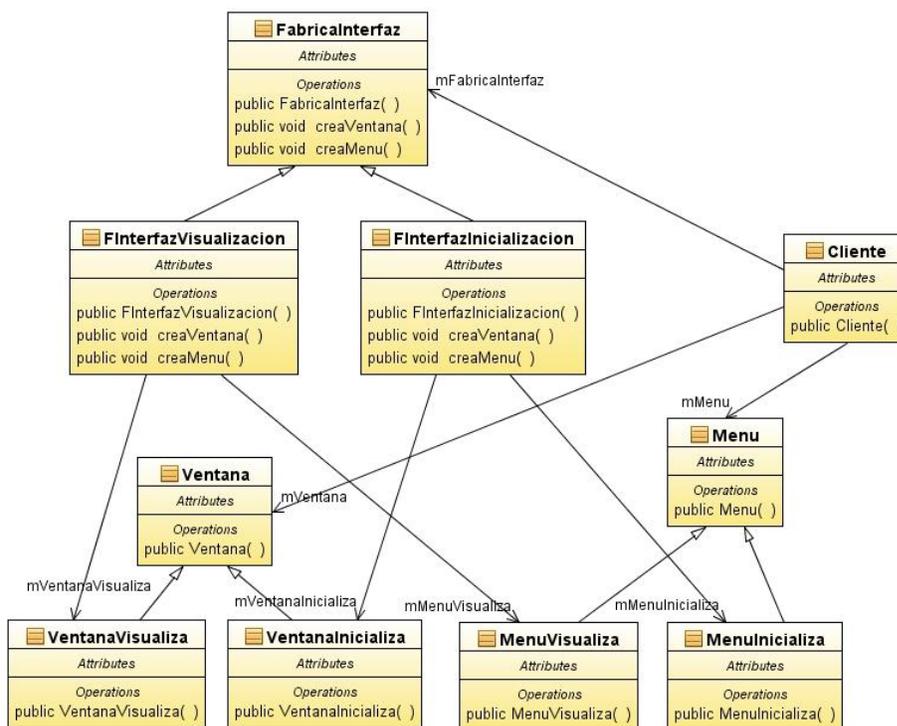
La Figura 8.32 exhibe el diagrama de clases que caracteriza el patrón *Abstract Factory*.



**Figura 8.32** El patrón Abstract Factory para creación de objetos productos requeridos por el comportamiento o por la lógica del sistema de software.

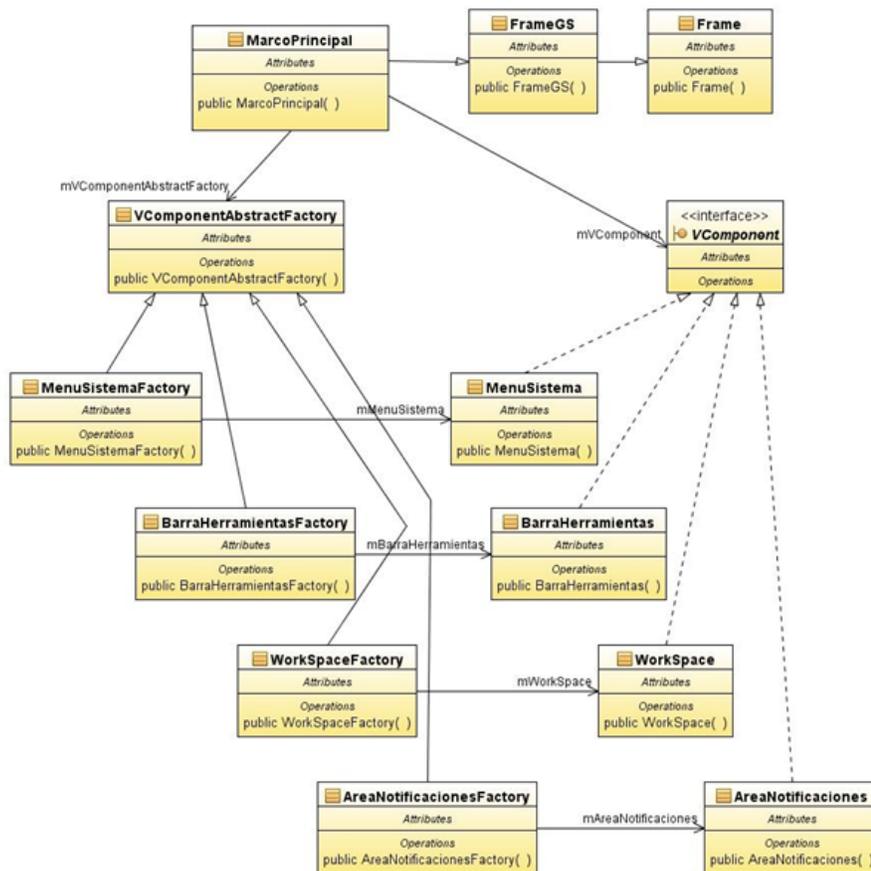
Como se puede apreciar en la Figura 8.32, la clase *Cliente* conoce la interfaz de las fábricas concretas (*FabricaConcreta1*, *FabricaConcreta2*) que producen los objetos-producto relacionados (*ProductoA1*, *ProductoA2*, *ProductoB1*, *ProductoB2*), a las cuales puede solicitar la creación del objeto-producto requerido, recibiendo éste una vez que ha sido elaborado. Nótese que la clase *Cliente* es compatible con la interfaz del objeto producto (*ProductoAbstractoA*, *ProductoAbstractoB*).

Las Figuras 8.33 y 8.34 ilustran ejemplos de uso del patrón *Abstract Factory* durante el diseño de sistemas de software. Como se puede apreciar en la Figura 8.33, en este caso el patrón *Abstract Factory* ha sido utilizado para crear objetos-producto gráficos que definen el comportamiento de un sistema de software orientado a eventos. La clase *Cliente* es compatible con la interfaz de las fábricas concretas (*FInterfazVisualizacion* y *FInterfazInicializacion*), las cuales producen productos del tipo ventana y menú (*VentanaVisualiza*, *VentanaInicializa*, *MenuVisualiza*, *MenuInicializa*). Una vez concluida la construcción de los objetos del tipo *Producto*, éstos son enviados a la clase *Cliente*, la cual conoce la interfaz de los mismos (*Ventana*, *Menu*). De esta forma, con el uso del patrón *Abstract Factory* se logró separar el comportamiento de la aplicación de la construcción de los objetos gráficos que intervienen en dicho comportamiento.



**Figura 8.33** Uso del patrón Abstract Factory para la creación de objetos-producto gráficos que definen el comportamiento de un sistema de software.

Por otra parte, en la Figura 8.34 ilustra el uso del patrón *Abstract Factory* en la construcción de componentes gráficos mucho más específicos, los cuales serán posteriormente ensamblados por la clase cliente. La clase cliente (*MarcoPrincipal*) conoce la interfaz (*VComponentAbstractFactory*) de las fábricas concretas (*MenuSistemaFactory*, *BarraHerramientasFactory*, *WorkSpaceFactory* y *AreaNotificacionesFactory*), a las cuales solicita la construcción de los componentes gráficos requeridos –*MenuSistema*, *BarraHerramientas*, *WorkSpace*, *AreaNotificaciones*– y al ser compatible con la interfaz de éstos (*VComponent*), los recibirá una vez que los mismos hayan sido construidos para proceder a su ensamblaje en un componente gráfico contenedor.



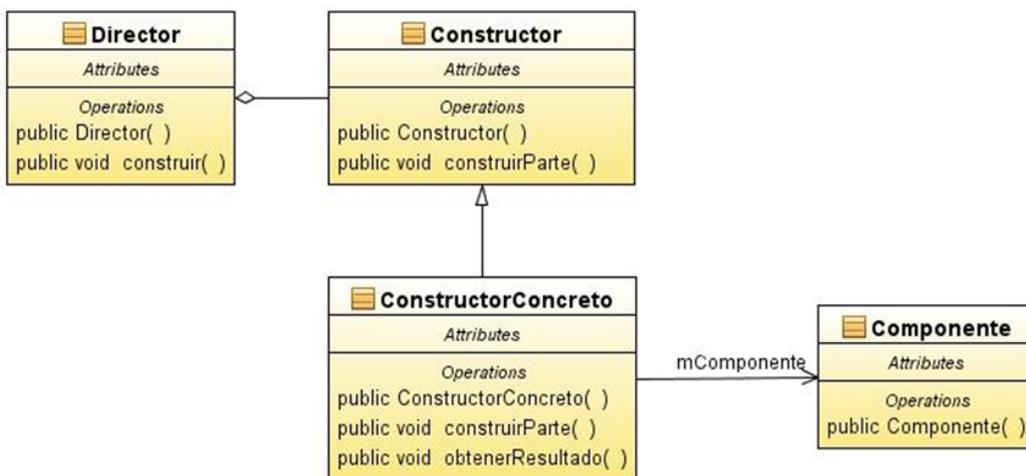
**Figura 8.34** Uso del patrón Abstract Factory para la creación de objetos-producto gráficos específicos que serán posteriormente ensamblados por la clase cliente. Este concepto de diseño pertenece al sistema RPS-GS, al cual ya nos hemos referido.

## Patrón Builder

El patrón *Builder* [2] proporciona una forma eficaz de separar la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones. El patrón *Builder* es fundamentalmente usado cuando:

- El algoritmo para crear un objeto complejo debe ser independiente de las partes de las cuales se compone dicho objeto y de cómo se ensamblan.
- El proceso de construcción debe permitir diferentes representaciones del objeto que está siendo construido.

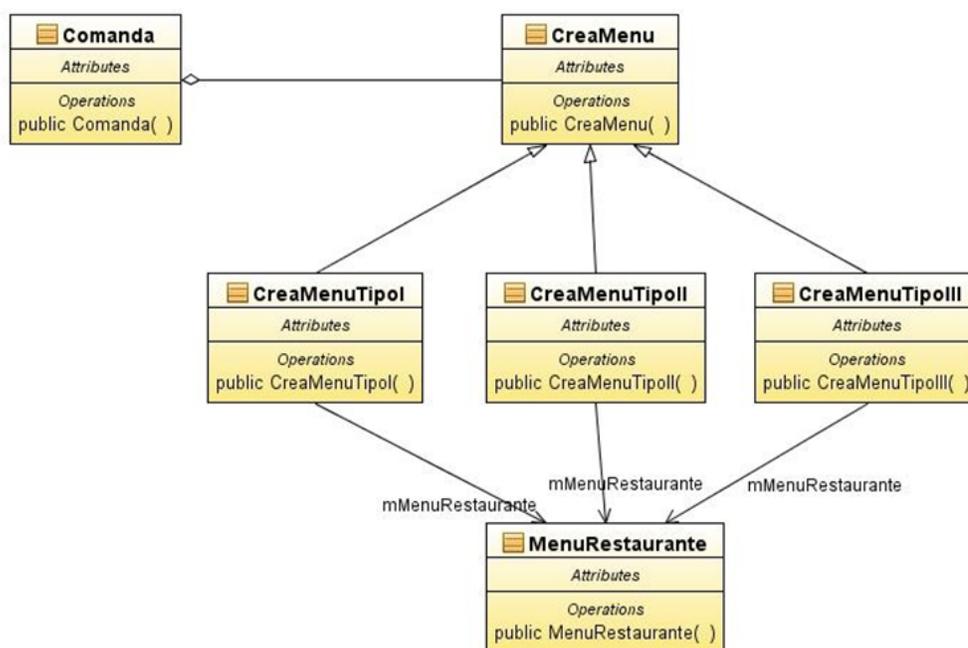
La Figura 8.35 muestra el diagrama de clases que caracteriza el patrón *Builder* en términos de los componentes y relaciones que lo integran. Nótese en esta figura que, aunque se ha representado un solo constructor concreto (*ConstructorConcreto*), comúnmente de la clase *Constructor* extienden y se especializan varios constructores concretos, de forma tal que cada uno de ellos sea capaz de construir el objeto-producto (*Componente*) con una determinada representación. La clase *Director* construye un *Componente* utilizando la interfaz *Constructor* y delegando la solicitud en el tipo de constructor (*ConstructorConcreto*) idóneo para construir el objeto con la representación requerida.



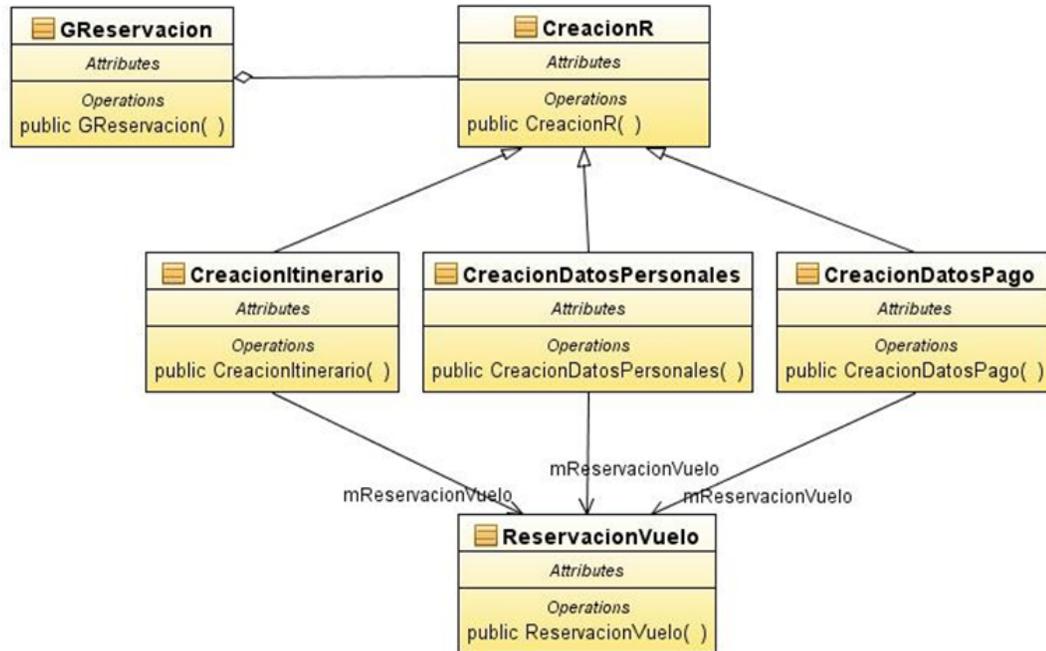
**Figura 8.35** El patrón *Builder* separa la construcción de un objeto de su representación.

Las Figuras 8.36 y 8.37 proporcionan ejemplos del uso del patrón *Builder*. En la Figura 8.36 se ilustra el uso del patrón *Builder* en el subsistema de gestión de comandas de un restaurante. La *Comanda* utiliza la interfaz *CreaMenu* para crear uno de entre tres tipos de menú, dependiendo del constructor al que le delegue la petición (*CreaMenuTipoI*, *CreaMenuTipoII* y *CreaMenuTipoIII*). El objeto-producto creado es *MenuRestaurante* y su representación dependerá del constructor concreto que lo construya. En la Figura 8.37 se muestra el uso del patrón *Builder* en

un fragmento del diseño del módulo *Reservaciones*, de un sistema de reservaciones de vuelos de una aerolínea. Como se puede apreciar en esta figura, el patrón *Builder* define tres constructores concretos (*CreacionItinerario*, *CreacionDatosPersonales*, *CreacionDatosPago*), donde cada uno de éstos o bien construye una representación específica del objeto-producto *ReservacionVuelo* la cual será enviada a la clase cliente *GReservacion* o bien participa en el ensamblaje del mismo. Esto último significa que el objeto producto resultante contendrá como parte de su estado el itinerario de vuelo, los datos personales del pasajero y los datos de pago. No obstante, la clase cliente *GReservacion* podría solicitar a la interfaz *CreacionR* una o cualquier combinación de representaciones del objeto-producto *ReservacionVuelo*.



**Figura 8.36** Uso del patrón *Builder* en el diseño de un sistema de gestión de comandas en un restaurante.



**Figura 8.37** Uso del patrón *Builder* en el diseño de un sistema de reservaciones de vuelos de una aerolínea.

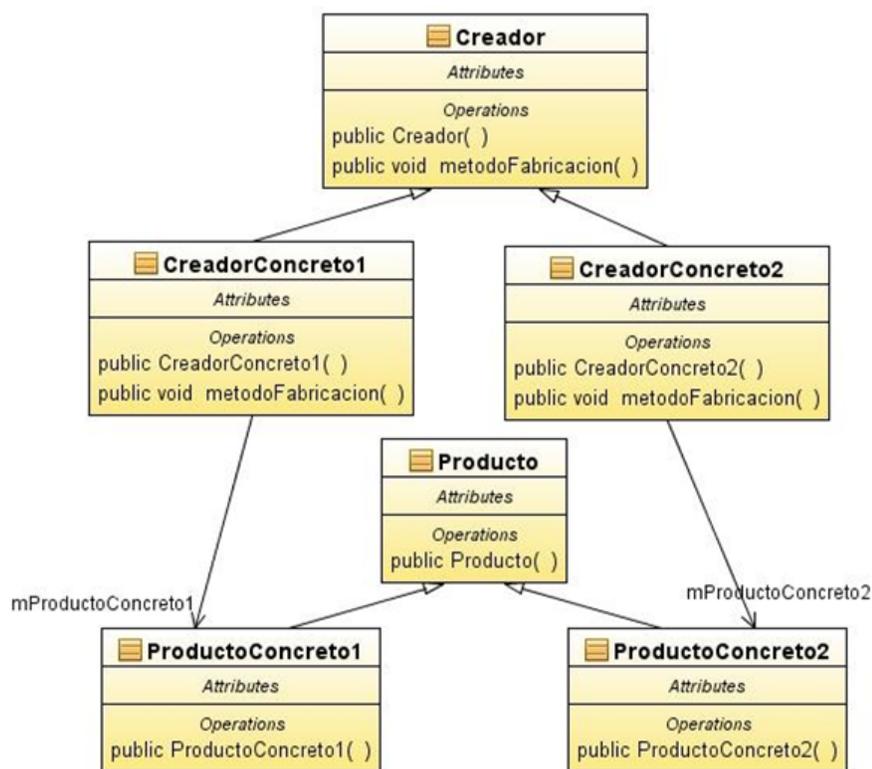
### Patrón *Factory Method*

El patrón *Factory Method* [2] podría verse como un caso particular del patrón *Abstract Factory*. Este patrón define una interfaz para crear un objeto, dejando que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos. Comúnmente, el patrón *Factory Method* es utilizado cuando:

- Una clase no puede prever la clase de objetos que debe crear.
- Una clase delega en sus subclases la especificación de los objetos que ésta crea.
- Las clases pueden delegar la responsabilidad en una de entre varias clases auxiliares.

La Figura 8.38 muestra el diagrama de clases que caracteriza al patrón *Factory Method*. Como se puede apreciar en esta figura, la superclase *Creador* delega en las subclases *CreadorConcreto1* y *CreadorConcreto2* la creación del objeto *Producto* requerido, ya sea *ProductoConcreto1* o *ProductoConcreto2*. Las Figuras 8.39 y 8.40 exhiben el uso del patrón *Factory Method* en un escenario al cual ya nos hemos referido en otras ejemplificaciones, el sistema de reservaciones de vuelos de una aerolínea. Como se muestra en la Figura 8.39, cualquier clase cliente compatible con la interfaz *CreaReservación* puede solicitar la creación tanto de un objeto *ReservacionVuelo* como de un objeto *ReservacionHotel*, siendo las subclases

*CreaReservacionVuelo* y *CreaReservacionHotel* las que deciden cual objeto-producto crear, es decir, un producto *ReservacionVuelo* o uno *ReservacionHotel*. De forma análoga, en el concepto de diseño ilustrado en la Figura 8.40, la clase cliente *GestorReservaciones* conoce la interfaz *CreadorReservaciones*, por lo que puede solicitar la creación de uno de tres tipos de objetos-producto: *ReservacionPrepago*, *ReservacionCargoTC* o *ReservacionFechaV*. En este caso, son nuevamente las subclases *CreadorReservacionPrepago*, *CreadorReservacionTC* y *CreadorReservacionFechaV* las que deciden cual objeto-producto construir.



**Figura 8.38** El patrón de creación *Factory Method*. A través de la jerarquía generalización-especialización, permite que una clase delegue en sus subclases la creación de objetos-producto.

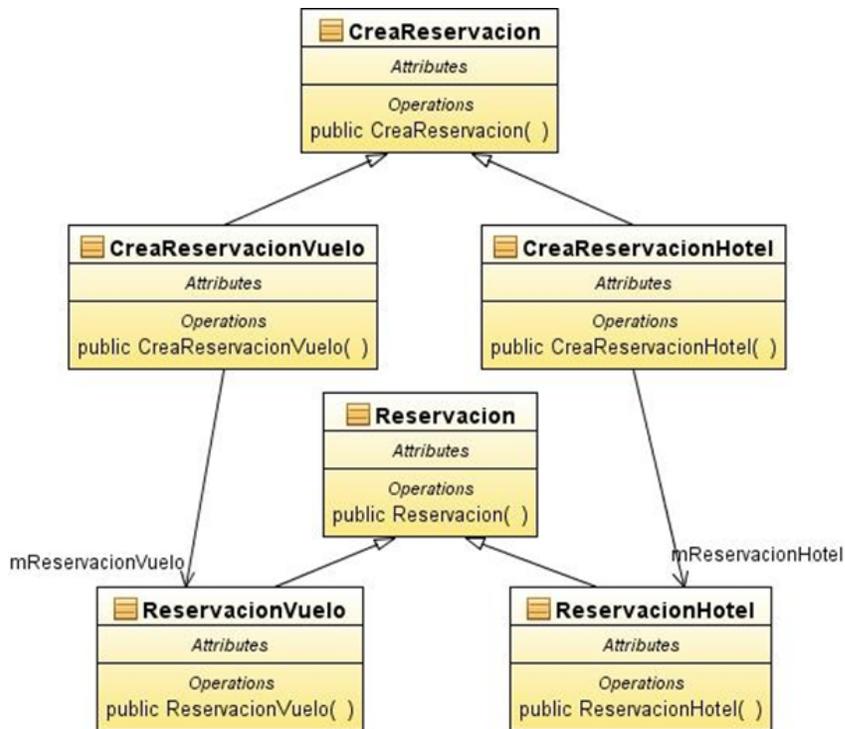


Figura 8.39 Uso del patrón de creación *Factory Method* en el diseño del módulo *Reservaciones* de un sistema de reservaciones de vuelo de una aerolínea.

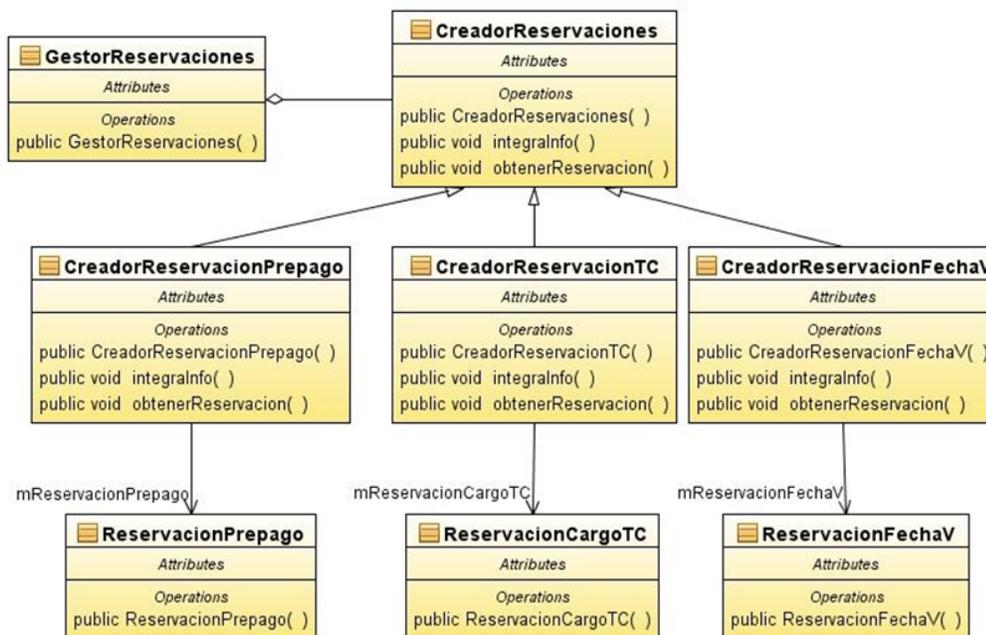


Figura 8.40 Uso del patrón de creación *Factory Method* en el diseño del módulo *Pago de Reservaciones* de un sistema de reservaciones de vuelo de una aerolínea.

## Patrón Prototype

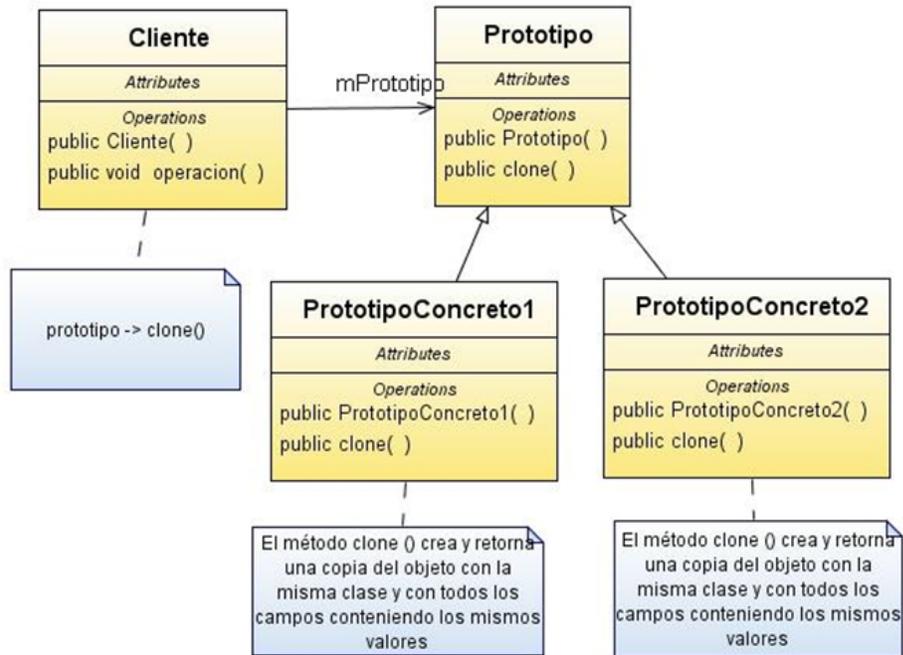
El patrón *Prototype* [2] permite crear nuevos objetos sin necesidad de instanciar nuevamente la clase base de la cual el objeto requerido es instancia, sino que el nuevo objeto se crea directamente copiando un objeto ya existente instancia de esta clase, y cuyo estado satisface los requerimientos del nuevo objeto a crear. En otras palabras, el patrón *Prototype*, especifica los tipos de objetos a crear por medio de una instancia prototípica (un objeto previamente creado y crea nuevos objetos copiando dicho prototipo). El patrón *Prototype* resulta de gran utilidad en las siguientes situaciones:

- Un sistema debe ser independiente de cómo se crean, se componen y se representan sus productos.
- Crear una instancia de una clase es un proceso complejo. Entonces, en lugar de crear varias instancias de la clase, se hacen copias de la instancia original (*Prototype*) modificando ésta según resulte.
- Se desea evitar construir una jerarquía de clases de fábricas paralela a la jerarquía de clases de los componentes.
- Cuando se desea reducir el número de clases (subclases).

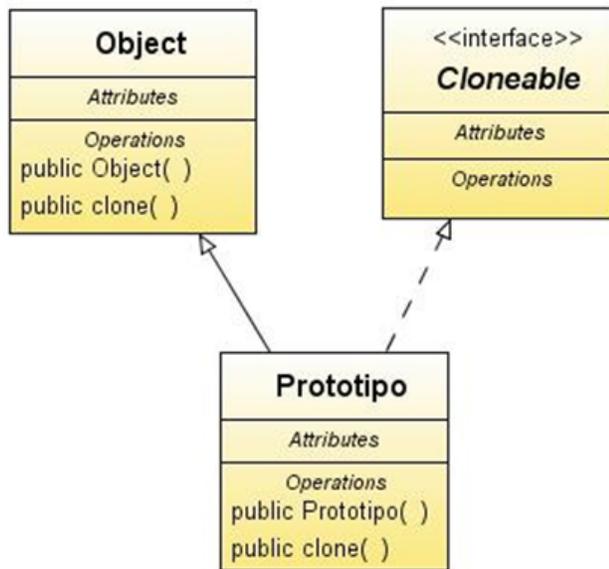
En la Figura 8.41 se muestra el diagrama de clases que describe las entidades y relaciones que intervienen en el patrón *Prototype*. Como se puede apreciar en esta figura, el objeto que se desea copiar (*PrototipoConcreto1*, *PrototipoConcreto2*) debo poner a disposición de la clase cliente (*Cliente*) un método que permita que dicho objeto sea copiado y devuelto a la clase cliente (método *clone()*).

Los lenguajes orientados a objetos proporcionan al menos un método para la creación de objetos a partir de la copia de los mismos. En el caso del lenguaje Java, está disponible el método *clone()* para la duplicación de objetos. El método *clone()* actúa como un constructor. Comúnmente, el método *clone()* al nivel de la subclase base invoca el método *clone()* de sus superclases para obtener la copia, hasta que éste alcanza el método *clone()* de la clase *Object*. El método *clone()* de la clase *Object* crea y retorna una copia del objeto con la misma clase y con todos los campos manteniendo los mismos valores.

Como se puede apreciar en la Figura 8.42, la clase del objeto a clonarse debe implementar la interfaz *Cloneable* para indicar al método *Object.clone()* que resulta legal para tal método crear una copia campo a campo de instancias de dicha clase. La clase *CloneNotSupportedException* se atraviesa para indicar que el método *clone()* en la clase *Object* ha sido llamado para clonar un objeto, pero que la clase de dicho objeto no implementa la interfaz *Cloneable*.



**Figura 8.41** El patrón de creación *Prototype* para la creación de objetos a través de la copia del objeto existente.



**Figura 8.42** La interfaz Cloneable Java debe ser implementada por la clase del objeto a ser clonado.

La Figura 8.43 ilustra un fragmento de diseño de un sistema bancario, donde ha sido utilizada la jerarquía generalización-especialización para definir todas las

clases que representan tipos de productos bancarios, tales como tarjeta de crédito. Como se aprecia en esta figura, hemos definido clases separadas para Tarjeta de Crédito con Tasa Estándar (*TarjetaCreditoTE*), Tarjeta de Crédito con Tasa Preferencial (*TarjetaCreditoTP*) y Tarjeta de Crédito con Tasa Exclusiva (*TarjetaCreditoTEX*). Sin embargo, en vez de definir todas estas clases, éstas podrían ser instancias de una única clase inicializadas con valores de tasas diferentes. Es decir, podríamos usar el patrón *Prototype* para reducir aún más el número de clases y evitar así instanciar tantas clases para la creación de objetos muy similares. La solución sería que *GestorCredito* cree un nuevo *ProductoBancario* copiando o clonando una instancia de una subclase de *ProductoBancario*, tal como *TarjetaCredito*. A esta instancia la llamaremos *prototype*. El diseño propuesto utilizando el patrón *Prototype* se muestra en la Figura 8.44.

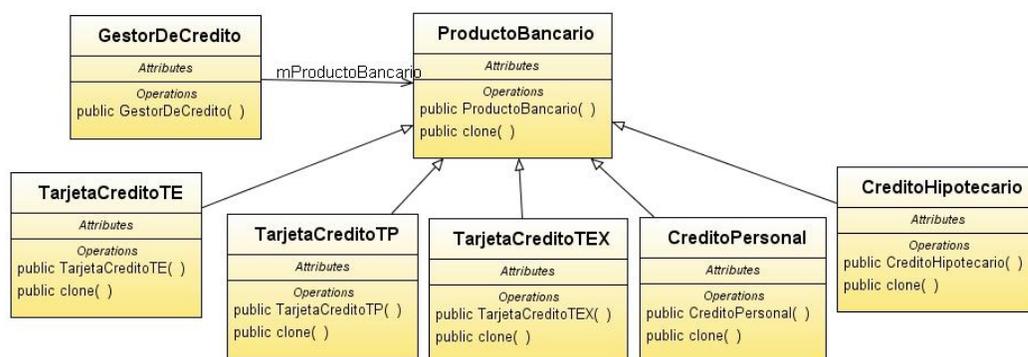


Figura 8.43 Fragmento de diseño de un sistema bancario.

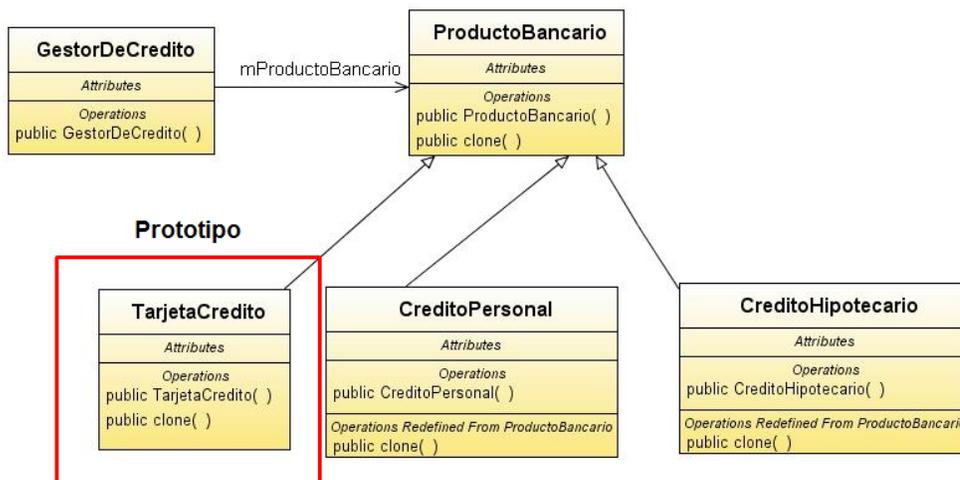


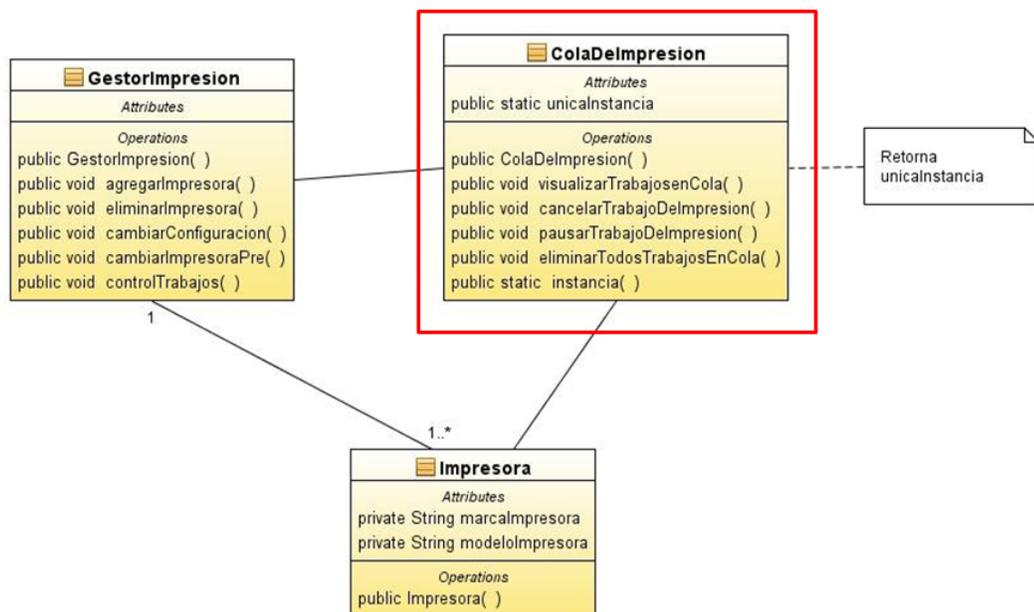
Figura 8.44 Fragmento de diseño de un sistema bancario en el cual se reduce el número de subclases utilizando el patrón *Prototype* (ver Figura 8.42).

## Patrón Singleton

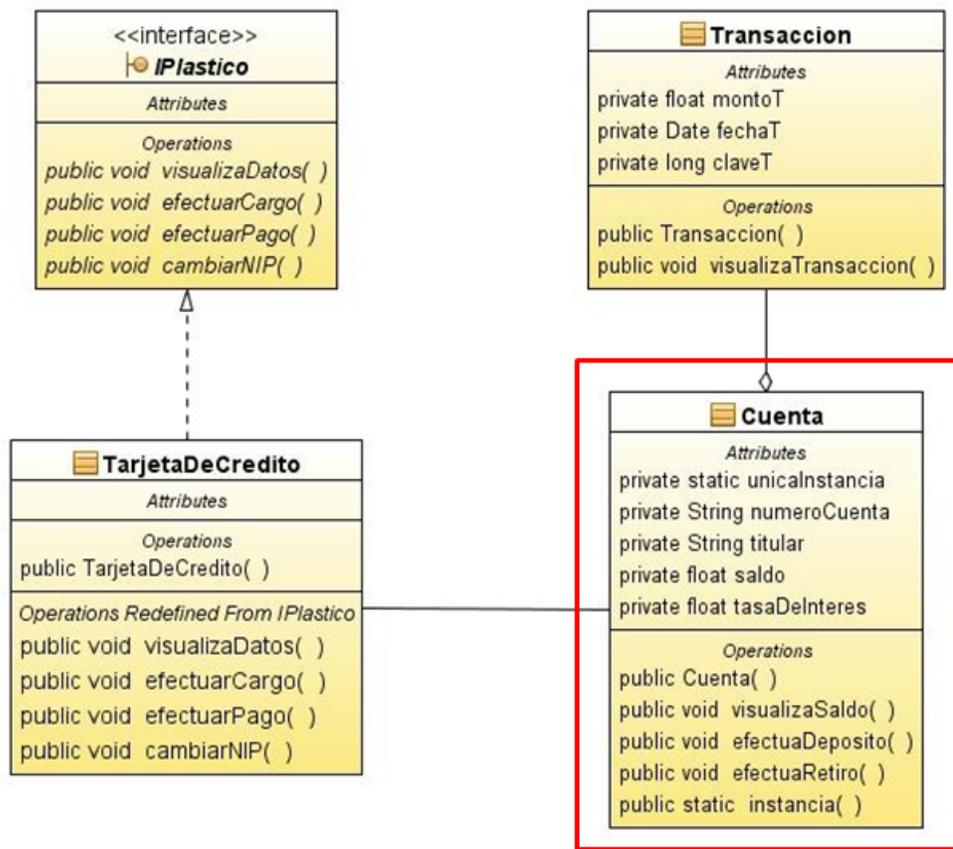
El patrón de creación *Singleton* [2] restringe la creación de objetos instancias de una clase, permitiendo que sólo se pueda instanciar un único objeto. Es decir, garantiza que una clase tenga a lo sumo una única instancia. Comúnmente, el patrón *Singleton* es utilizado cuando:

- Debe haber exactamente una instancia de una clase, y ésta debe ser accesible a los clientes desde un punto de acceso conocido.
- La única instancia puede ser extendida a través de la herencia, por lo que los clientes podrían utilizar la instancia extendida sin modificar su código.

En el patrón *Singleton*, la propia clase es la responsable de su única instancia. La clase debe garantizar que no se pueda crear ninguna otra instancia. Una de las variantes más directas para utilizar el patrón *Singleton*, es cuando declaramos la clase estática (*static*), de forma tal que no sea posible instanciar objetos de la misma, fungiendo la propia clase como su única instancia. En Java una clase estática es aquella en la cual todos sus métodos son declarados como estáticos, como por ejemplo la clase *Math*, la cual no permite instancias de la misma. A diferencia de los restantes patrones de diseño que hemos presentado, donde para cada uno de éstos se ha proporcionado un diagrama de clases que permita comprender la estructura del mismo, el patrón *Singleton* no requiere de dicho diagrama de clases, ya que su papel se comprende mucho mejor a través de ejemplos de aplicación, tal como se ilustra en las Figuras 8.45 y 8.46.



**Figura 8.45** Uso del patrón *Singleton* en el diseño de un componente de impresión.



**Figura 8.46** Uso del patrón *Singleton* en el diseño de un componente de transacciones en un sistema de pago con tarjetas de crédito.

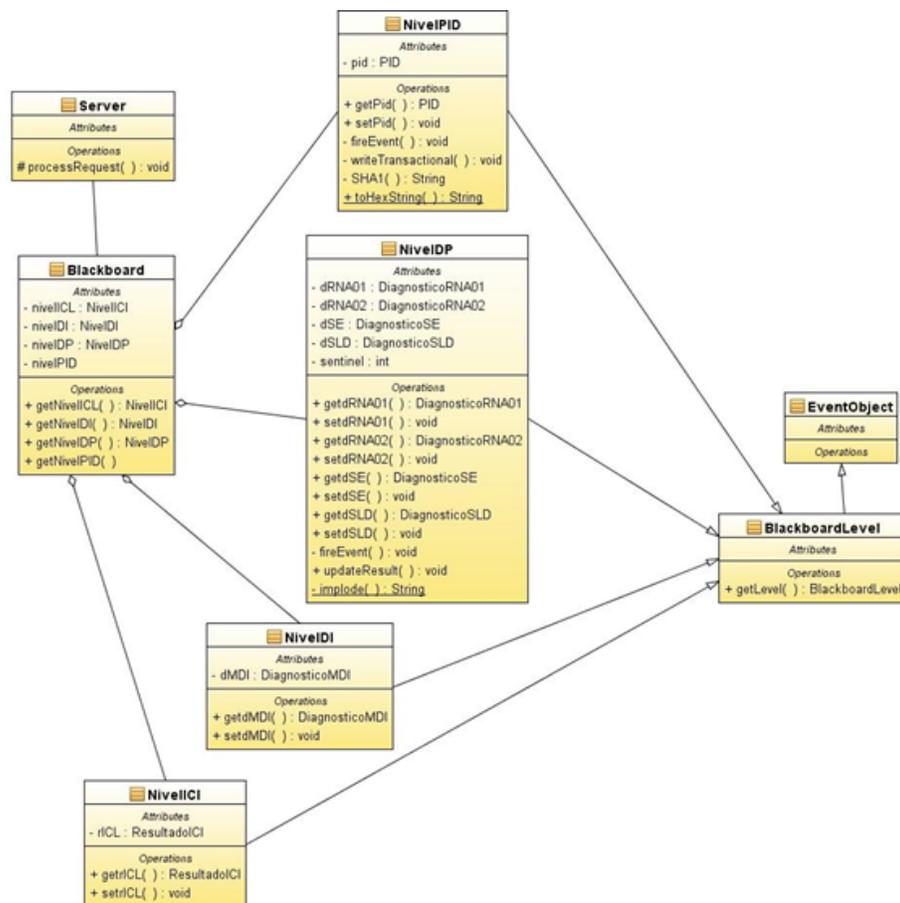
Como se puede apreciar en la Figura 8.45, el patrón *Singleton* ha sido utilizado como parte del diseño de un componente de impresión, donde se debe garantizar que de la clase *ColaDeImpresion* solo se permita crear única instancia, la cual será utilizada por los clientes *GestorImpresion* e *Impresora*. Por otra parte, en la Figura 8.46 se ilustra el uso del patrón *Singleton* en el diseño de un componente de transacciones de un sistema de pagos con tarjetas de crédito. En este caso, el patrón *Singleton* garantiza que solo haya una instancia de la clase *Cuenta*, de forma tal que cualquier objeto cliente que requiera de los servicios del objeto instancia de *Cuenta* hagan referencia al mismo y único objeto.

## VIII.4 Casos de estudio

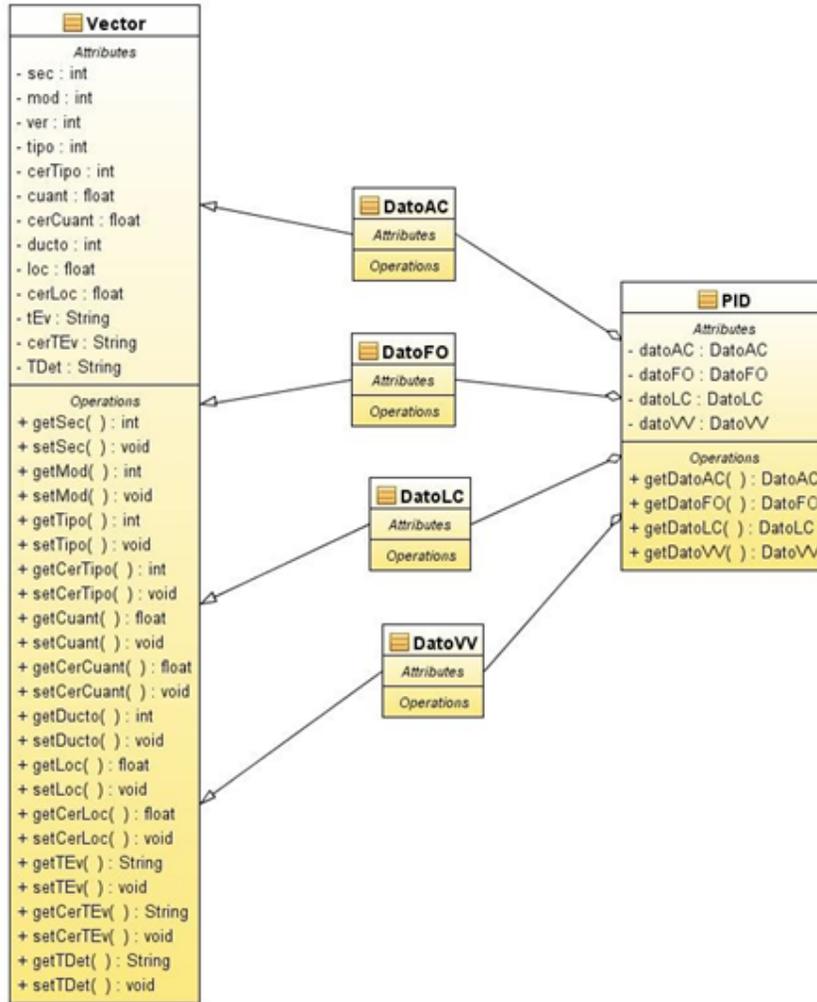
### VIII.4.1 Sistema de detección de fugas en ductos que transportan hidrocarburos. modularización y diseño de componentes

Durante las fases de diseño del subsistema SPI, componente del sistema de detección de fugas en ductos que transportan hidrocarburos, se utilizaron diferentes

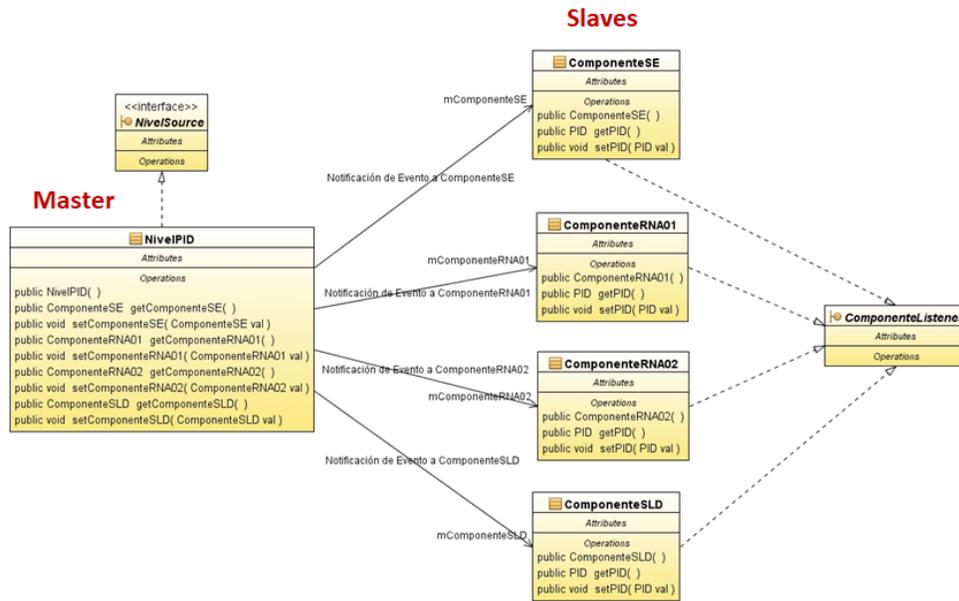
patrones de diseño para modelar aspectos relacionados con la creación de objetos, la estructura del sistema, la organización del trabajo y el control de acceso. En las Figuras 8.47 a la 8.55 se ilustra cómo fueron utilizados dichos patrones.



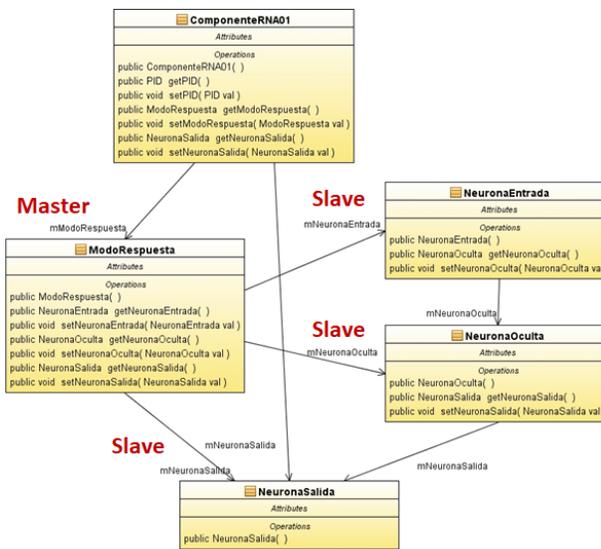
**Figura 8.47** Uso del patrón *Composite* en el diseño de la pizarra compartida (*Blackboard*) en el subsistema SPI. La clase *Blackboard* es una agregación de los niveles que la conforman: *NivelPID*, *NivelDP*, *NivelDI* y *NivelICI*.



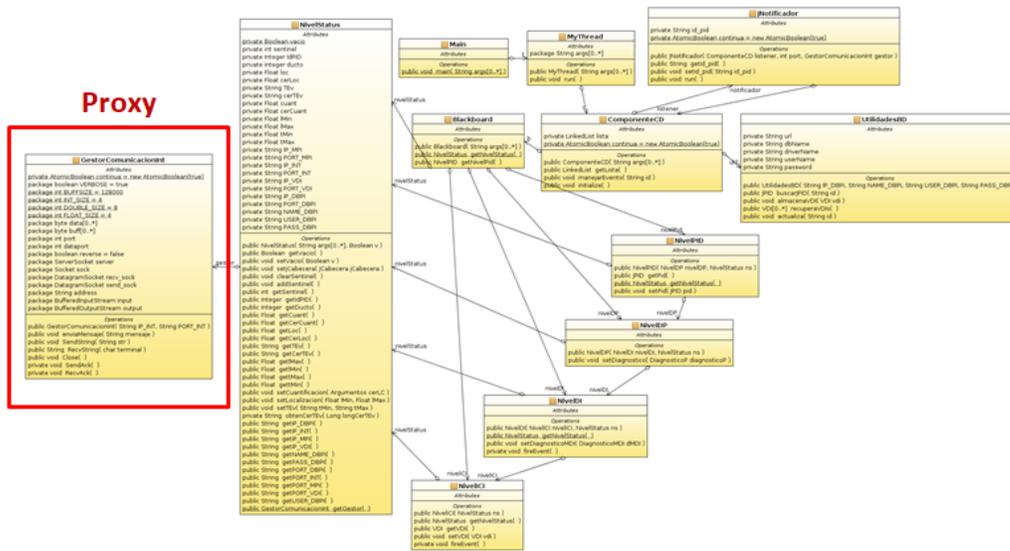
**Figura 8.48** Uso del patrón *Composite* en el diseño del componente *PID* (Patrón Integral de Datos) en el subsistema SPI. La clase *PID* es una agregación de los diferentes tipos de datos que la conforman: *DatoAC*, *DatoFO*, *DatoLC* y *DatoVV*.



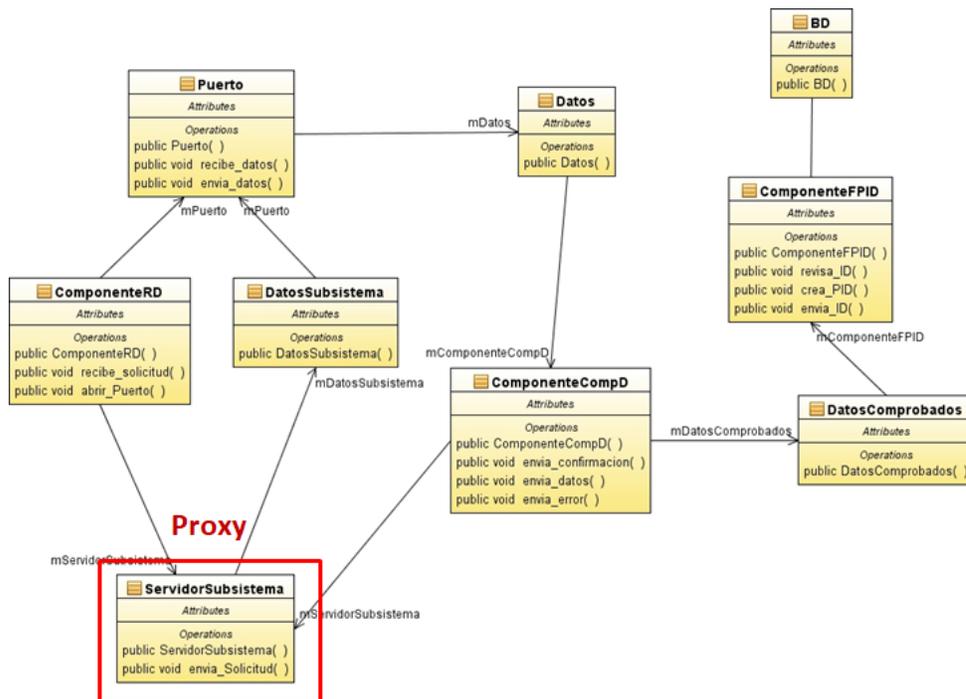
**Figura 8.49** Uso del patrón *Master-Slave* en el diseño de las relaciones entre la pizarra compartida (*Blackboard*) y los componentes que operan sobre la misma en el subsistema SPI. La clase *NivelPID* del *Blackboard* funge como *Master*, mientras que las clases *ComponenteSE*, *ComponenteRNA01*, *ComponenteRNA02* y *ComponenteSLD* se desempeñan como los *Slaves* concurrentes.



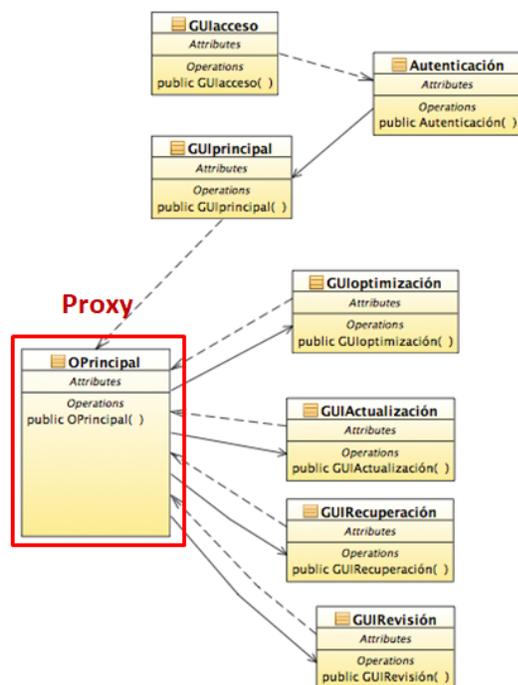
**Figura 8.50** Uso del patrón *Master-Slave* en el diseño de las relaciones entre el componente de generalización de la red neuronal (*ModoRespuesta*) y los componentes en base a los cuales se estructura la red neuronal (*NeuronaEntrada*, *NeuronaOculta* y *NeuronaSalida*). La clase *ModoRespuesta* funge como *Master*, mientras que las clases *NeuronaEntrada*, *NeuronaOculta* y *NeuronaSalida* se desempeñan como los *Slaves*.



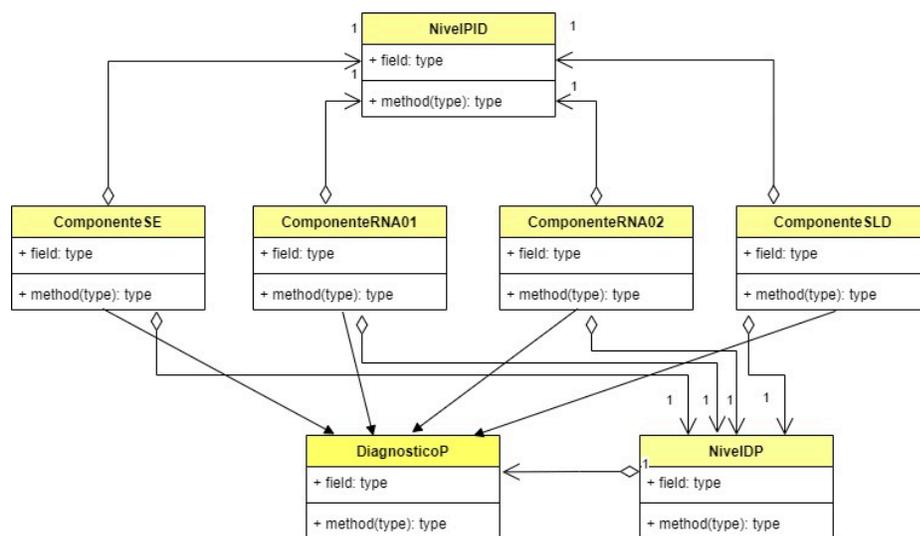
**Figura 8.51** Uso del patrón *Proxy* como interfaz lógica (*GestorComunicacionInt*) para mediar la comunicación entre las clases clientes y la clase *NivelStatus*. De esta forma, las clases clientes delegan a la clase proxy *GestorComunicacionInt* sus peticiones sobre el status de un determinado nivel del *Blackboard*.



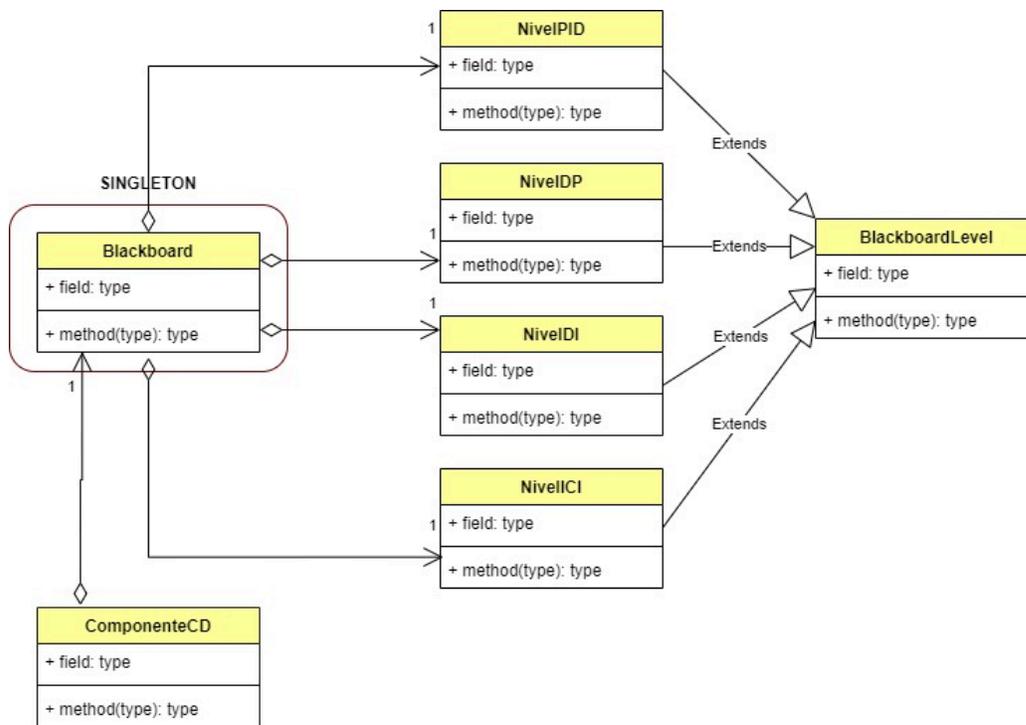
**Figura 8.52** Uso del patrón *Proxy* como interfaz lógica (*ServidorSubsistema*) para mediar la comunicación entre las clases clientes (*ComponenteRD* y *ComponenteCompD*) y la clase *Servidor* (no incluida en el diagrama).



**Figura 8.53** Uso del patrón *Proxy* en el componente monolítico del SPI. En este caso, la clase *Proxy* representa un oyente genérico de los eventos que ocurren un en conjunto de GUI estrechamente relacionadas entre sí. La clase *Proxy* *OPrincipal* funge como intermediaria entre las GUI y las clases de la lógica que pueden procesar las peticiones efectuadas a través de dichas GUI.



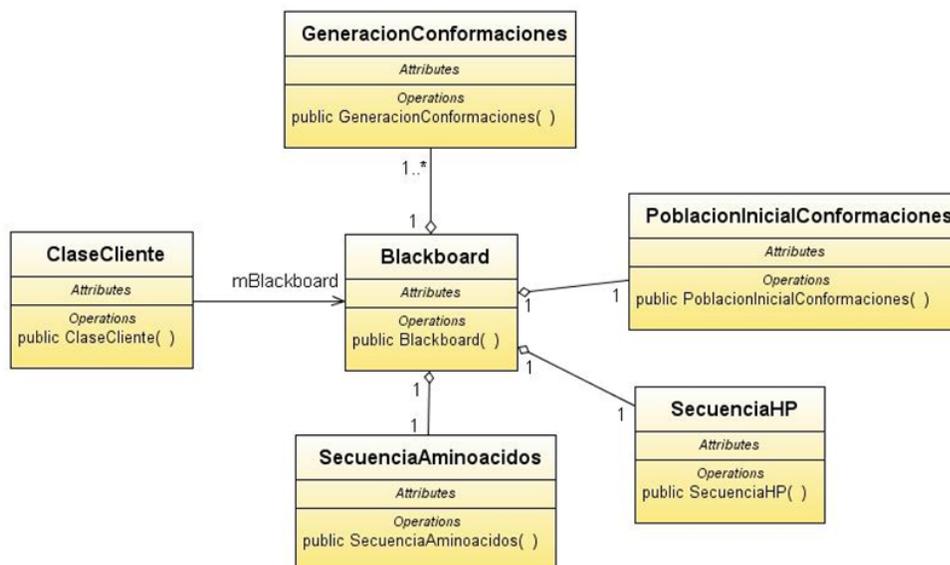
**Figura 8.54** Uso del patrón *Builder* para el diseño de la relación entre los constructores concretos (*ComponenteSE*, *ComponenteRNA01*, *ComponenteRNA02* y *ComponenteSLD*) y el componente construido *DiagnosticoP*. Cada uno de los cuatro constructores concretos define la representación a crear del producto *DiagnosticoP*.



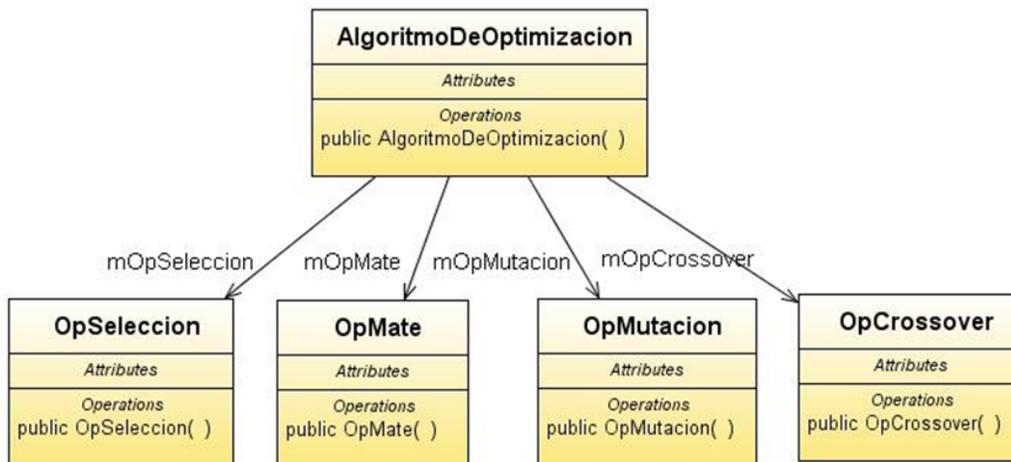
**Figura 8.55** Uso del patrón *Singleton* en el diseño de la clase *Blackboard* para garantizar que sólo exista una única instancia de la misma. De esta forma, ninguna de las clases que conocen la interfaz de la clase *Blackboard* podrá crear otra instancia de la misma, preservándose así un único estado del objeto instancia de dicha clase.

### VIII.4.2 Plataforma bioinformática Evolution

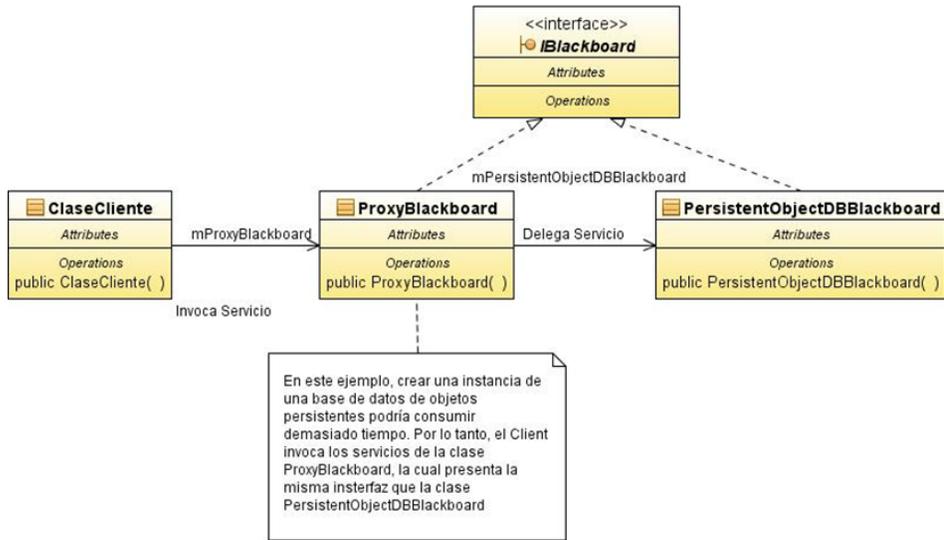
El diseño de la plataforma bioinformática Evolution también consideró el uso de algunos patrones de diseño para modelar aspectos afines con la estructura del sistema, la organización del trabajo, el control de acceso y la creación de objetos. En las Figuras 8.56 a la 8.59 se ilustra cómo fueron utilizados dichos patrones.



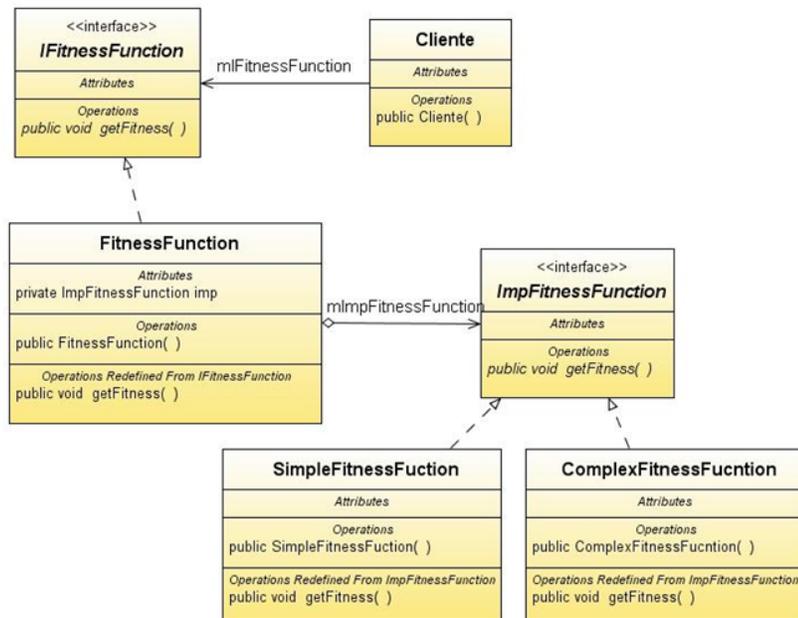
**Figura 8.56** Uso del patrón *Composite* en el diseño de la pizarra compartida (*Blackboard*) en la plataforma bioinformática *Evolution*. La clase *Blackboard* es una agregación de los niveles que la conforman: *SecuenciaAminoacidos*, *SecuenciaHP*, *PoblacionInicialConformaciones* y *GeneracionConformaciones*.



**Figura 8.57** Uso del patrón *Master-Slave* en el diseño de las relaciones entre la clase *AlgoritmoDeOptimizacion* quien funge como *Master* y las clases que representan los operadores genéticos *OpSeleccion*, *OpMate*, *OpMutacion* y *OpCrossover*, las cuales juegan el papel de *Slave*. En este caso, los *Slaves* son invocados en orden secuencial y no concurrente. La clase *Master* delega en los *Slaves* las operaciones de selección, formación de parejas, cruce y mutación.



**Figura 8.58** Uso del patrón *Proxy* para delegar a una clase intermediaria la creación de una instancia de un objeto persistente, el cual es serializado y deserializado. De esta forma, el cliente *ClaseCliente* evita la interacción directa con la instancia de la clase *PersistentObjectDBBlackboard*, lo cual podría consumir mucho tiempo.



**Figura 8.59** Uso del patrón *Bridge* para desacoplar la abstracción *ImpFitnessFunction* de las dos posibles implementaciones proporcionadas para la misma: *SimpleFitnessFuction* y *ComplexFitnessFuction*. De esta forma, se evita un enlace permanente entre la abstracción y su implementación, por lo que la implementación puede cambiar en tiempo de ejecución.

## VIII.5 Referencias del capítulo

1. Grand, M. Patterns in Java. Volume 1 - A catalog of reusable design patterns illustrated with UML. Wiley computer Publishing. 2002.
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. 1998.
3. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. Pattern-Oriented Software Architecture. A System of Patterns. Wiley. 2001
4. Zlobin, G. *Learning Python Design Patterns*. Packt Publishing. 2013.
5. Cusumano, M.A. The Factory Approach to Large-Scale Software Development: Implications for Strategy, Technology, and Structure. Classic Reprints Series, 2017.
6. Garland, J., Anthony, R. Large Scale Software Architecture. A Practical Guide Using UML. Wiley, 2003.
7. Janakiram, D. Building Large Scale Software Systems. McGraw Hill Education, 2013.
8. Lakos, J. Large-Scale C++ Software Design. Addison Wesley, 1996.
9. Screerer, A. Coordination in Large-Scale Agile Software Development: Integrating Conditions and Configurations in Multiteam Systems (Progress in IS). Springer, 2017.

# Capítulo IX Gestión de la Configuración

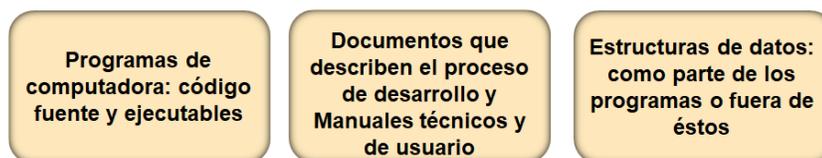
En el desarrollo de sistemas de software, y en particular en el desarrollo de software a gran escala, la gestión de la configuración es en sí una fase más del proceso de desarrollo, la cual tiene una gran importancia en la gestión de la evolución de un sistema de software [6-10]. La gestión de la configuración [1-5] del sistema de software se refiere al desarrollo y aplicación de estándares y procedimientos para gestionar la evolución, versiones y cambios de un sistema de software. Ésta inicia cuando comienza el proceso de desarrollo de software y concluye cuando el software ha sido liberado.

La gestión de la configuración del sistema de software trata y controla principalmente:

- Los cambios de los requerimientos durante el desarrollo del sistema y su incorporación en nuevas versiones del sistema de software.
- Los cambios de los requerimientos, correcciones de errores, adaptaciones para hardware, etc., que se incorporan en cada versión.

## IX.1 Identificación de los elementos de la configuración del sistema

La información resultante de un proceso de ingeniería de software puede ser agrupada en tres principales categorías, tal como se ilustra en la Figura 9.1. Es precisamente en estas tres categorías donde podemos identificar los elementos de la configuración del sistema.



**Figura 9.1** Documentos y productos resultantes de un proceso de ingeniería de software.

Los elementos de la configuración del sistema se refieren a todos los documentos y productos generados a través de las diferentes fases de desarrollo del sistema software, tales como:

- El plan del proyecto de software.
- La especificación del sistema de software.
- La especificación de los requerimientos.
- La especificación del diseño: diseño arquitectónico y diseño detallado.
- Las estructuras de datos.
- Los listados de código fuente.
- Los conjuntos de datos de prueba.
- Los manuales de operación, de instalación y de usuario.

La gestión de la configuración del sistema de software propone procedimientos y estándares de forma tal que se garanticen la gestión de cambios y la gestión de versiones.

## IX.2 Gestión de cambios

La principal fuente de cambios, tanto durante el desarrollo de un sistema de software como durante el tiempo de vida de dicho sistema, viene dada por el cambio de los requerimientos funcionales. Los requerimientos funcionales de un sistema de software pueden cambiar según las nuevas expectativas y necesidades de los clientes, lo cual debe conllevar a una gestión de control de cambios, para valorar y determinar si el cambio solicitado procede y de ser el caso, proceder a la ejecución del mismo en forma correcta.

Cuando la fuente de cambios viene dada por los requerimientos del sistema de software, entonces comúnmente el cliente debe trabajar con el equipo encargado de la gestión de la configuración del sistema, para evaluar el impacto del cambio y decidir su realización. Otra fuente de cambios viene dada por errores descubiertos durante la fase de pruebas del sistema de software.

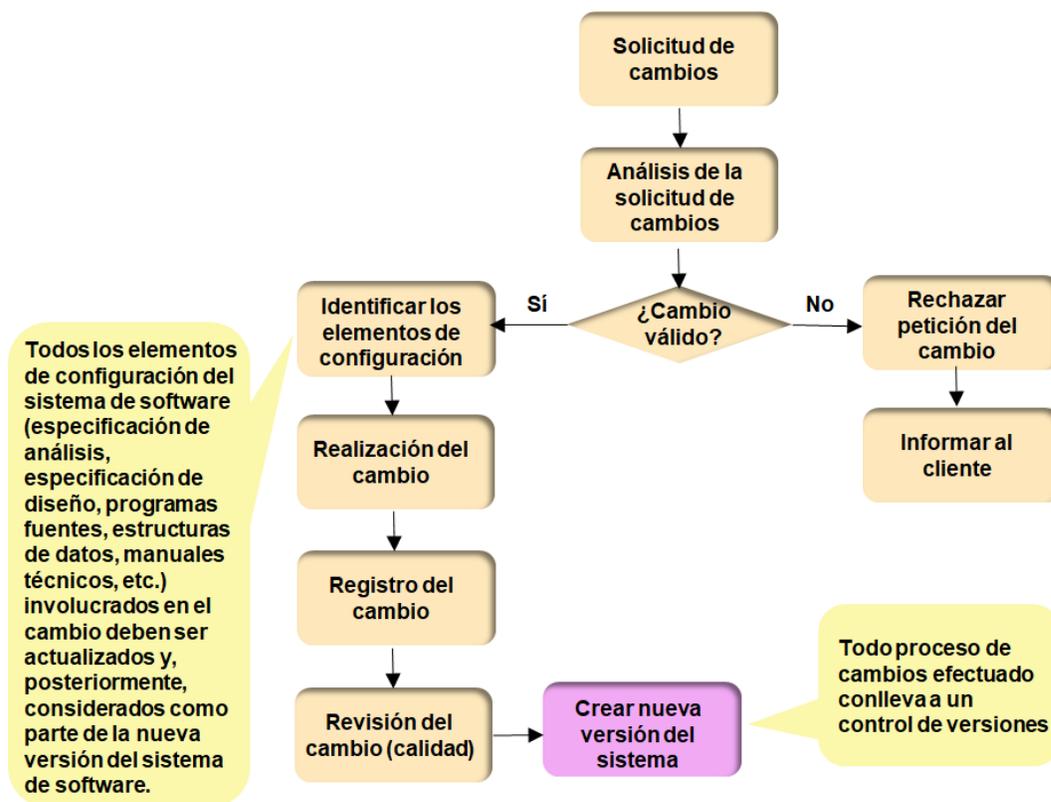
La gestión de cambios (también referida como control de cambios) se basa en el uso de procedimientos humanos y herramientas automáticas, que proporcionan un mecanismo para la evaluación del cambio solicitado, y de ser el caso, la realización, seguimiento y revisión del mismo.

Los procedimientos del control de cambios incluyen el análisis de costos y los beneficios de los cambios propuestos. A través de estos procedimientos se decide cuáles cambios merecen la pena llevarse a cabo, y se identifican los componentes y elementos de configuración del sistema de software que deben cambiar.

La gestión de cambios, como parte de la gestión de la configuración del sistema software, abarca las siguientes actividades:

- Identificación del cambio.
- Control del cambio.
- Implementación del cambio.
- Información del cambio a todas las partes afectadas.

La Figura 9.2 ilustra el flujo de actividades envueltas en la gestión de cambios.



**Figura 9.2** El proceso de gestión de cambios en ingeniería del software.

La complejidad de los sistemas de software a gran escala, exige que ante todo cambio solicitado por el cliente, se ejecute de forma rigurosa el proceso de control de cambios. En un sistema de software a gran escala, toda gestión de cambios ejecutada debe conllevar a una gestión de versiones.

### IX.3 Gestión de Versiones

La gestión de versiones se refiere al uso de procedimientos y herramientas para gestionar las versiones de los elementos de configuración del sistema de software tales como: la especificación de los requerimientos, la especificación del diseño arquitectónico, la especificación del diseño detallado, las estructuras de datos, los programas fuentes, etc. Como ya habíamos mencionado, toda gestión de cambios conlleva a una gestión de versiones.

La gestión de versiones (también referida como control de versiones) conlleva a la identificación y al mantenimiento de los registros de las diferentes versiones del sistema de software, gestionando los siguientes aspectos:

- Durante el proceso de desarrollo de software, es imprescindible garantizar que las diferentes versiones del sistema se puedan recuperar cuando éstas sean requeridas.
- Por otra parte, es necesario garantizar que estas versiones no se modifiquen, intercambien o confundan, de forma accidental o errónea, por parte del equipo de desarrollo.

Una versión de un sistema de software es una descripción, instancia o copia de dicho sistema, que difiere de alguna forma de otras descripciones. Los aspectos que pueden conllevar a la diferencia entre versiones de un sistema de software son múltiples, aunque podemos mencionar los siguientes:

- Cambios en la funcionalidad o requerimientos.
- Cambios en las estructuras de datos.
- Cambios en la arquitectura.
- Cambios en las interfaces gráficas de usuario.
- Cambios en la configuración de hardware.

En el desarrollo de sistemas de software a gran escala, el control de versiones es un proceso complejo, dado el elevado número de componentes de software que integran el sistema, y que cada uno de estos componentes podría contar con un número diferente de versiones. Para crear una versión particular de un sistema de software a gran escala, es necesario especificar la versión de cada uno de los componentes que debe formar parte de esta versión del sistema.

Entre las principales técnicas utilizadas para la identificación de los componentes de un sistema de software, se encuentran las siguientes:

- Identificación basada en la numeración de versiones.
- Identificación basada en atributos.

- Identificación orientada al cambio.

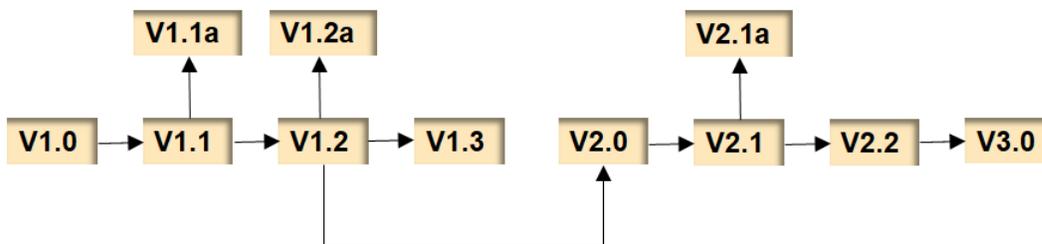
### IX.3.1 Identificación de componentes basada en la numeración de versiones

Cuando se usa la técnica de numeración de versiones, a cada componente se le asigna un número de versión explícito y único. Ésta es la técnica de identificación de componentes más utilizada. Al nombre del componente (o del sistema) se le añade un número explícito y único de versión. Por ejemplo:

- Evolution 4.2
- Cellulat 3.4
- SPI 2.1

La primera versión se identifica como **1.0**, y las subsiguientes versiones se identifican como **1.1**, **1.2**, y así sucesivamente. Un cambio o conjunto de cambios notable, conlleva a que se salte a una nueva versión, por ejemplo, **2.0**, y a partir de aquí se reinicia la numeración de las versiones en **2.1**, **2.2**, y así sucesivamente. No necesariamente cada versión previa de un sistema debe dar lugar a una versión sucesiva. La derivación de versiones no es necesariamente un proceso lineal.

Como se ilustra en la Figura 9.3, la representación de la técnica numeración de versiones, se basa en gráfico de derivación de versiones, compuesto por **rectángulos (o cuadrados o círculos)** y **flechas**. Un **rectángulo (cuadrado o círculo)** representa una versión, mientras que una **flecha** representa una transición entre dos versiones. Es decir, una **flecha** va dirigida desde una **versión fuente** hasta una **nueva versión** del sistema de software creada a partir de la **versión fuente**.



**Figura 9.3** Representación de la técnica “numeración de versiones”, para la identificación de los componentes de un sistema de software.

### IX.3.2 Identificación de componentes basada en atributos

La técnica numeración de versiones, aun siendo la más comúnmente utilizada para el control de versiones, presenta el problema que el nombre explícito de la versión no indica los diferentes atributos considerados en la propuesta/desarrollo de dicha versión, y que permitirían identificar con mucho más detalle una versión.

La técnica de control de versiones, identificación basada en atributos, se basa precisamente en identificar las versiones considerando un conjunto de atributos de identificación y valores para dichos atributos.

Cuando se usa la técnica identificación basada en atributos, los principales atributos que permiten identificar las versiones son los siguientes:

- El lenguaje de desarrollo.
- Estado o fase del desarrollo.
- Plataforma de hardware.
- Tecnología predominante.
- Fecha de creación.

El control de versiones basado en la técnica de identificación de atributos facilita mucho el desarrollo de nuevas versiones a partir de cualquiera de las versiones existentes. Cuando se utiliza esta técnica, las versiones se identifican y se recuperan utilizando un conjunto único de valores de los atributos.

### IX.3.3 Identificación de componentes orientada al cambio

La técnica identificación orientada al cambio está mucho más orientada a la identificación de la versión del sistema de software de forma global, que a la identificación individual de los componentes que integran el sistema. Cada nueva versión del sistema de software tiene asociado un conjunto de cambios, el cual describe los cambios efectuados a los componentes del sistema, que fueron requeridos para implementar el cambio que encierra la nueva versión.

En primer lugar, se identifican los conjuntos de cambios y posteriormente se van generando las nuevas versiones a partir de estos conjuntos de cambios. Comúnmente, una nueva versión del sistema corresponde a un conjunto de cambios particular.

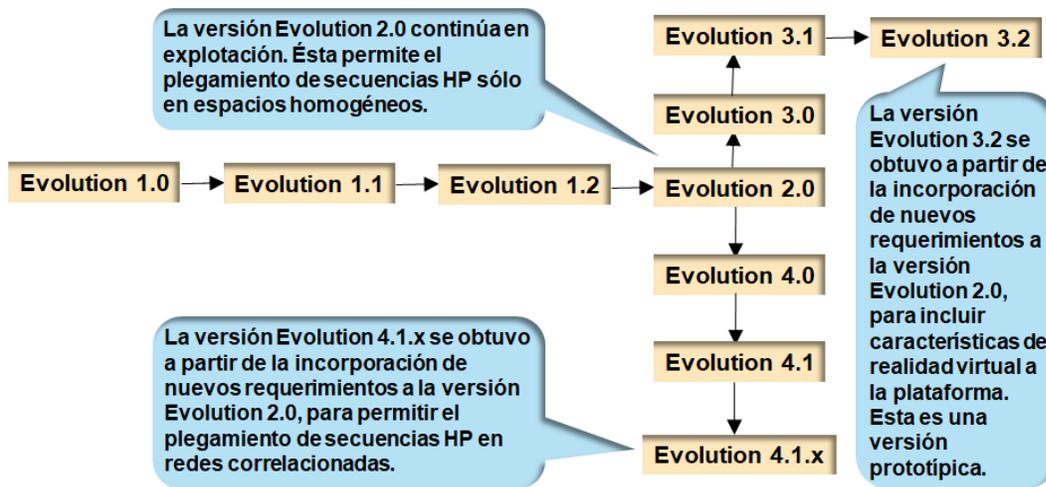
Para aplicar los diferentes conjuntos de cambios a un sistema, y generar así las nuevas versiones, ante todo se debe garantizar la consistencia y compatibilidad de la secuencia de aplicación de los conjuntos de cambios. En otras palabras, no se pueden aplicar a un sistema de software conjuntos de cambios que sean incompatibles entre sí.

## IX.4 Caso de Estudio

### IX.4.1 Plataforma computacional Evolution. un laboratorio virtual para el modelado y simulación del plegamiento de proteínas. control de cambios y control de versiones

#### IX.4.1.1 Control de versiones en la plataforma bioinformática Evolution utilizando la identificación de componentes basada en la numeración de versiones

En la Figura 9.4 se ilustra el control de versiones llevado a cabo durante el desarrollo de la plataforma bioinformática Evolution, utilizando la identificación de componentes basada en la numeración de versiones.



**Figura 9.4** Estructura de derivación de versiones en la plataforma bioinformática Evolution.

#### IX.4.1.2 Control de versiones en la plataforma bioinformática Evolution utilizando la identificación de componentes basada en atributos

En la Figura 9.5 se puede apreciar el control de versiones llevado a cabo durante el desarrollo de la plataforma bioinformática Evolution, utilizando la identificación de componentes basada en atributos.

- ❑ Evolution (lenguaje = “Java”, fase de desarrollo = “prototipo funcional”, arquitectura = “MVC monolítica”, modelo predominante = “modelo HP, algoritmos evolutivos”, tecnología predominante = “java 2D/3D”, fecha = “diciembre 2012”)
- ❑ Evolution (lenguaje = “Java”, fase de desarrollo = “producto liberado”, arquitectura = “MVC monolítica”, modelo predominante = “modelo HP, algoritmos evolutivos, redes correlacionadas”, tecnología predominante = “java 2D/3D”, fecha = “septiembre 2014”)
- ❑ Evolution (lenguaje = “Go”, fase de desarrollo = “prototipo funcional”, arquitectura = “MVC cliente-servidor”, modelo predominante = “modelo HP, algoritmos evolutivos”, tecnología predominante = “Realidad Virtual”, fecha = “octubre 2016”)

**Figura 9.5** Identificación de componentes basada en atributos, en la plataforma bioinformática Evolution.

#### IX.4.1.3 Control de versiones en la plataforma bioinformática Evolution utilizando la identificación de componentes orientada al cambio

En la Figura 9.6 se muestra el control de versiones llevado a cabo durante el desarrollo de la plataforma bioinformática Evolution, utilizando la identificación de componentes orientada al cambio.

- ❑ Evolution versión 4.1.x, que satisface el siguiente conjunto de cambios:
  - Inclusión del espacio de plegamiento 2D en espacios no homogéneos (Redes Correlacionadas).
  - Incorporación de las funcionalidades para la selección de la red correlacionada 2D particular y del área de dicha red en la cual se efectuará el plegamiento.
  - Incorporación del algoritmo de plegamiento 2D en espacios no homogéneos.
  - Incorporación de las funcionalidades para la visualización del plegamiento resultante en redes correlacionadas 2D.

**Figura 9.6** Identificación de componentes orientada al cambio, en la plataforma bioinformática Evolution.

## IX.5 Referencias del capítulo

1. Pfleeger, S. L. Ingeniería de software: Teoría y práctica. Pearson Education, 2002.
2. Pressman, R. S. Ingeniería del software: Un enfoque práctico. McGraw-Hill, 2010.
3. Sommerville, I. Ingeniería del software. Pearson Addison Wesley, 2012.
4. Tsui, F., Karam, O., Bernal, B. Essentials of software engineering. Jonas & Bartlett Learning books, 2015.
5. Van Vliet, H. Software engineering: Principles and practice. John Wiley & Sons, 2000.
6. Cusumano, M.A. The Factory Approach to Large-Scale Software Development: Implications for Strategy, Technology, and Structure. Classic Reprints Series, 2017.
7. Garland, J., Anthony, R. Large Scale Software Architecture. A Practical Guide Using UML. Wiley, 2003.
8. Janakiram, D. Building Large Scale Software Systems. McGraw Hill Education, 2013.
9. Lakos, J. Large-Scale C++ Software Design. Addison Wesley, 1996.
10. Screerer, A. Coordination in Large-Scale Agile Software Development: Integrating Conditions and Configurations in Multiteam Systems (Progress in IS). Springer, 2017.

# Capítulo X Gestión de la Calidad y las Pruebas

## X.1 Introducción

La gran complejidad inherente al desarrollo de los proyectos de software a gran escala (funcionalidad extensa y compleja, dependencia de una gran variedad de requerimientos no funcionales, grandes requerimientos de procesamiento de información, necesidad de alta modularización y desacoplamiento, integración de diferentes componentes de software, hardware y comunicaciones, entre otras características), hace de la gestión de la calidad una actividad crucial a desarrollarse de forma transversal a través de todas las etapas del desarrollo de este tipo de sistemas [1-5].

Calidad del software [6-10] significa que:

- El sistema software desarrollado satisface plenamente los requerimientos funcionales especificados, así como las expectativas y necesidades del cliente.
- Tanto el diseño como la implementación corresponden de manera precisa a los requerimientos funcionales especificados.
- Los requerimientos no funcionales especificados –tales como, portabilidad, seguridad, eficiencia, reusabilidad y extensibilidad, etc.– han sido considerados durante el desarrollo del sistema software.
- El sistema software desarrollado es correcto, eficiente, flexible, confiable y seguro.
- La reusabilidad y extensibilidad del sistema software desarrollado han sido garantizadas.
- La calidad ha sido asegurada (ha sido medida y controlada) a través de cada una de las fases de desarrollo del sistema software: análisis de requerimientos, diseño arquitectónico, diseño de componentes, implementación, etc.

En resumen, entre los principales atributos de calidad del software, se encuentran aquellos que ya identificamos antes como requerimientos no funcionales o restricciones:

- Fiabilidad
- Flexibilidad
- Seguridad
- Robustez
- Adaptabilidad
- Portabilidad
- Modularidad
- Usabilidad
- Reutilización
- Protección
- Eficacia

Una de las principales razones que ha impactado significativamente en los últimos años en el mejoramiento de la calidad en el desarrollo de software a gran escala, ha sido la adopción de nuevas técnicas y tecnologías tales como:

- El desarrollo iterativo e incremental con signos visibles del producto de software en desarrollo. Las metodologías de desarrollo ágil de software.
- El desarrollo orientado a objetos, que garantiza reusabilidad y extensibilidad, y, por lo tanto, un desarrollo orientado al cambio.
- El soporte de herramientas CASE en varias de las etapas de desarrollo del software (planeación y estimación, diseño, implementación, etc.).

Como se puede apreciar en la Figura 10.1, la gestión de la calidad del software abarca tres actividades fundamentales:

1. Garantía de la calidad.
2. Planificación de la calidad.
3. Control de la calidad.

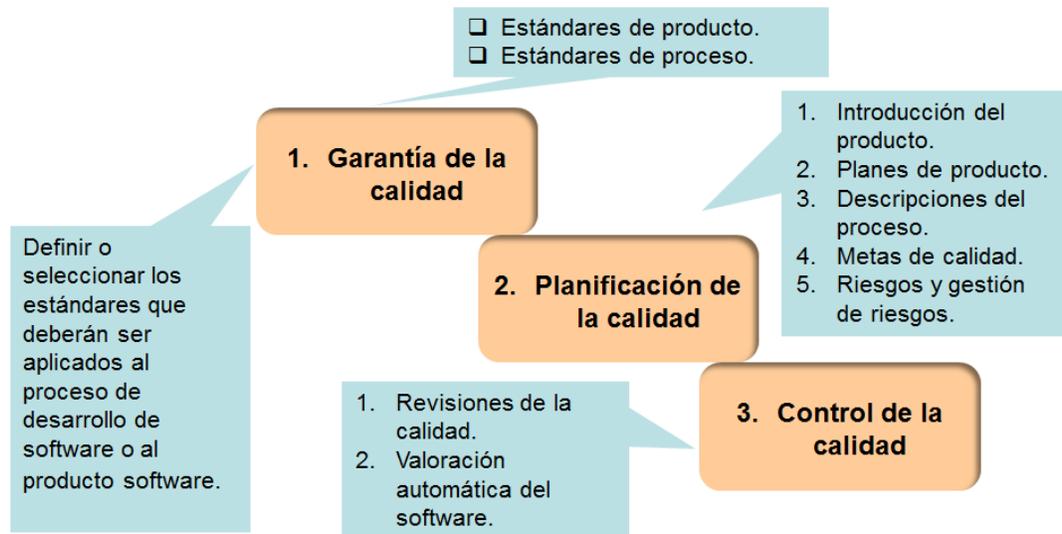


Figura 10.1 Principales actividades en la gestión de la calidad del software.

## X.2 Pruebas del software

El objetivo de la fase de pruebas es encontrar en la medida de lo posible, lo que el software no cumple, es decir, en cierta forma es una fase destructiva, es decir, durante las pruebas se busca detectar lo malo que hay en el software [11]. Cuando se diseñan las pruebas que deberán realizarse, se debe descartar la idea de que el Software está bien hecho. Por esto, no es recomendable que las mismas personas que diseñan el Software sean las que diseñen sus pruebas. El proceso de pruebas del software inicia con la *planeación de las pruebas*, después se hace el *diseño de pruebas* y finalmente éstas se *ejecutan*. A lo largo de todo el proceso se lleva a cabo el *control* de las pruebas.

En la *planeación* se define el plan global de las pruebas: el plan de pruebas (agenda de las actividades de prueba, objetivos y requerimientos de las pruebas, personal que va a hacer las pruebas, recursos SW y HW), riesgos y definición de las herramientas a utilizar. Durante el *diseño* se identifican los elementos que deben ser probados. Se especifican las condiciones y datos de prueba en base a los elementos a probar. Se diseña la estructura de las pruebas y el ambiente que requieren. Las pruebas se estructuran, se organizan por grupos y categorías, y se les asignan prioridades en base a un análisis de riesgos. Durante la *ejecución* se llevan a cabo las pruebas especificadas, observando y reportando los resultados. Además, se lleva a cabo un *control* durante el diseño y ejecución de las pruebas el cual tiene como objetivos: monitorear el progreso, medir y analizar los resultados, realizar acciones correctivas y tomar decisiones estratégicas.

### X.3 Gestión de las pruebas

Durante el desarrollo de software a gran escala, la gestión de pruebas constituye una actividad crucial a desarrollarse no solo sobre el producto software, sino también de forma transversal a través de todas las etapas del desarrollo del sistema de software.

Las pruebas del software consisten en un conjunto de actividades encaminadas a identificar posibles fallas, errores, omisiones, etc., tanto durante el desarrollo del producto software como durante la ejecución del mismo. Éstas deben abarcar cada una de las fases de desarrollo del producto software, y no centrarse solamente en la ejecución del software, ya que en cada una de éstas es posible detectar fallas, errores, omisiones, etc. Las pruebas del software pueden mostrar la presencia de errores, pero no pueden garantizar la ausencia de los mismos.

El ciclo de desarrollo de pruebas del software abarca pruebas relacionadas con:

- El proceso de desarrollo:
  - Recolección de requerimientos.
  - Análisis de los requerimientos.
  - Diseño arquitectónico.
  - Diseño de componentes.
  - Implementación.
- El producto software:
  - Los componentes del sistema software.
  - El sistema software.

Con relación al producto software, las pruebas se clasifican en dos categorías principales:

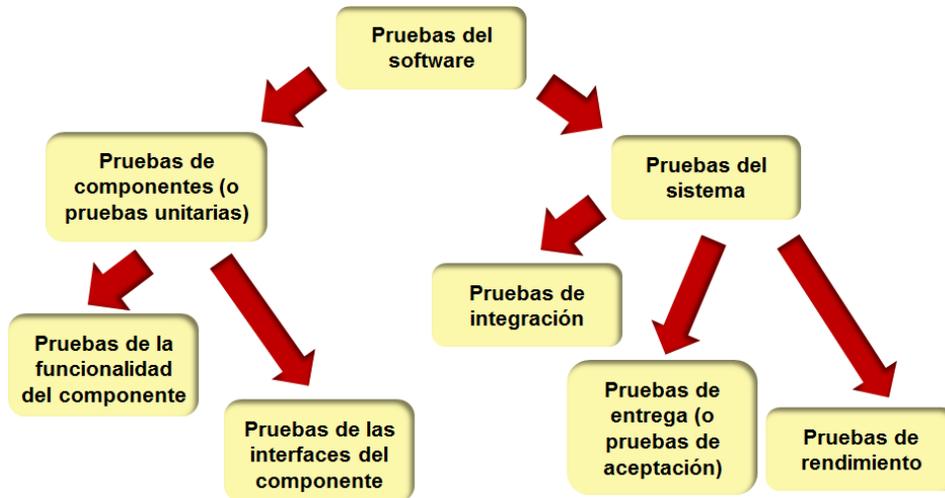
- Pruebas de componentes o pruebas unitarias:
  - Pruebas de la funcionalidad del componente.
  - Pruebas de las interfaces del componente.
- Pruebas del sistema:
  - Pruebas de integración.
  - Pruebas de entrega o pruebas de aceptación.
  - Pruebas de rendimiento.

La Tabla 10.1 y la Figura 10.2 proporcionan una descripción de estos dos tipos de pruebas del software.

**Tabla 10.1** Tipos de Pruebas del Software.

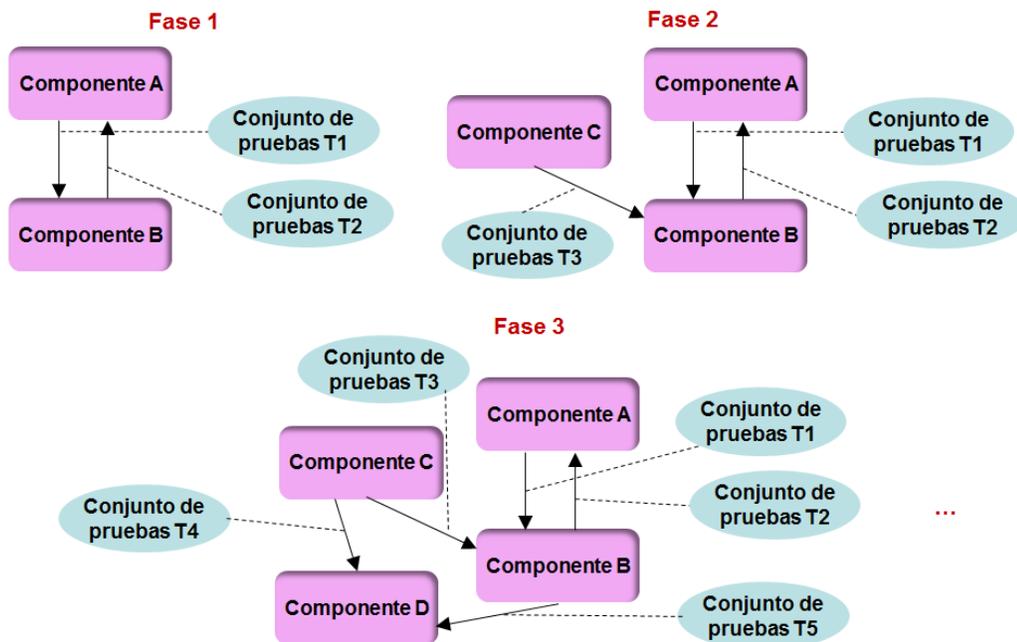
<b>Tipo de prueba</b>	<b>Descripción</b>
<b>Pruebas de componentes o pruebas unitarias</b>	<p>Es el proceso encaminado a la identificación de posibles errores, defectos, fallos u omisiones en los componentes individuales del sistema. En el enfoque orientado a objetos, los diferentes tipos de componentes que deben probarse son los siguientes:</p> <ul style="list-style-type: none"> <li>• Métodos formando parte de un objeto.</li> <li>• Clases de objetos con sus atributos y métodos.</li> <li>• Componentes formados por diferentes objetos.</li> </ul>
<b>Pruebas de la funcionalidad del componente</b>	<p>Se ocupan de probar que el componente de software realmente satisfaga la funcionalidad esperada del mismo, identificando cualquier tipo de error, defecto, fallo u omisión en su funcionamiento.</p>
<b>Pruebas de las interfaces del componente</b>	<p>Se ocupan de probar que la interfaz del componente compuesto de software se comporte de acuerdo a sus especificaciones. Un componente compuesto está integrado por varios objetos que interactúan entre sí.</p>
<b>Pruebas del sistema</b>	<p>Las pruebas del sistema toman lugar en la medida en que se van integrando los componentes que forman parte del sistema. Cada vez que dos o más componentes son integrados, se debe probar si el funcionamiento del sistema resultante (sistema integrado) es el esperado.</p>

<p><b>Pruebas de integración</b></p>	<p>Las pruebas de integración se ocupan principalmente de identificar posibles errores, defectos, fallos u omisiones en el sistema integrado resultante. Cada vez que un nuevo componente viene integrado, el sistema resultante debe ser probado. El principal problema que debe ser gestionado durante las pruebas de integración, es la localización de los errores. Un procedimiento de integración incremental facilitará mucho la localización de los errores.</p>
<p><b>Pruebas de entrega o pruebas de aceptación</b></p>	<p>Las pruebas de entrega o pruebas de aceptación se ocupan de probar una entrega del sistema que será liberada al cliente. Las pruebas de entrega se concentran en probar la funcionalidad del sistema. Es decir, si dada una entrada particular, se obtiene la salida que debe producir dicha funcionalidad. Este tipo de pruebas no se ocupan de la implementación del sistema, solo de su funcionalidad.</p>
<p><b>Pruebas de rendimiento</b></p>	<p>Las pruebas de rendimiento se llevan a cabo una vez que el sistema ha sido integrado completamente y las pruebas de entrega han sido superadas. Las pruebas de rendimiento se concentran en atributos de calidad o requerimientos no funcionales tales como: fiabilidad, flexibilidad, robustez, seguridad, adaptabilidad, portabilidad, etc. Las pruebas de rendimiento se ejecutan a partir de la construcción de “perfiles operacionales” o escenarios que reflejan la combinación real de trabajo que debería ser procesada por el sistema.</p>

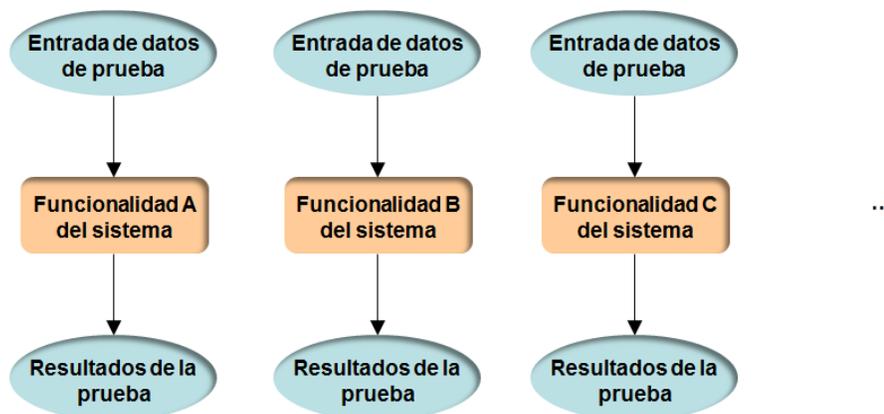


**Figura 10.2** Tipos de Pruebas del Software.

La Figura 10.3 ilustra el procedimiento que caracteriza las pruebas de integración incrementales. Por otra parte, la Figura 10.4 muestra el procedimiento característico de las pruebas de entrega o de aceptación.



**Figura 10.3** Pruebas del Sistema. Procedimiento de Integración Incremental.



**Figura 10.4** Pruebas del Sistema. Procedimiento de Pruebas de Entrega o de Aceptación.

## X.4 Caso de estudio

En este epígrafe ilustraremos algunas de las actividades llevadas a cabo durante la etapa “Gestión de Pruebas” de la plataforma bioinformática Evolution. De forma particular, mostraremos algunas de las actividades correspondientes a las pruebas de componentes y a las pruebas de integración.

### X.4.1 Pruebas de componentes (Pruebas Unitarias) en la plataforma computacional Evolution.

Como ya se indicó antes, las pruebas de componentes o pruebas unitarias se refieren a la identificación de posibles errores, defectos, fallos u omisiones en los componentes individuales del sistema. Cuando se sigue un enfoque orientado a objetos, como el utilizado en el diseño e implementación de la plataforma bioinformática Evolution, los componentes a probar abarcan: los métodos formando parte de un objeto, las clases de objetos con sus atributos y métodos, y los componentes formados por un grupo de objetos.

La Tabla 10.2 relaciona los componentes, clases y métodos que implementan los operadores de selección, cruce y mutación en el algoritmo genético de la plataforma bioinformática Evolution. La funcionalidad de cada método, clase y componente relacionados en esta tabla fue verificada y probada durante la Gestión de Pruebas de Componentes.

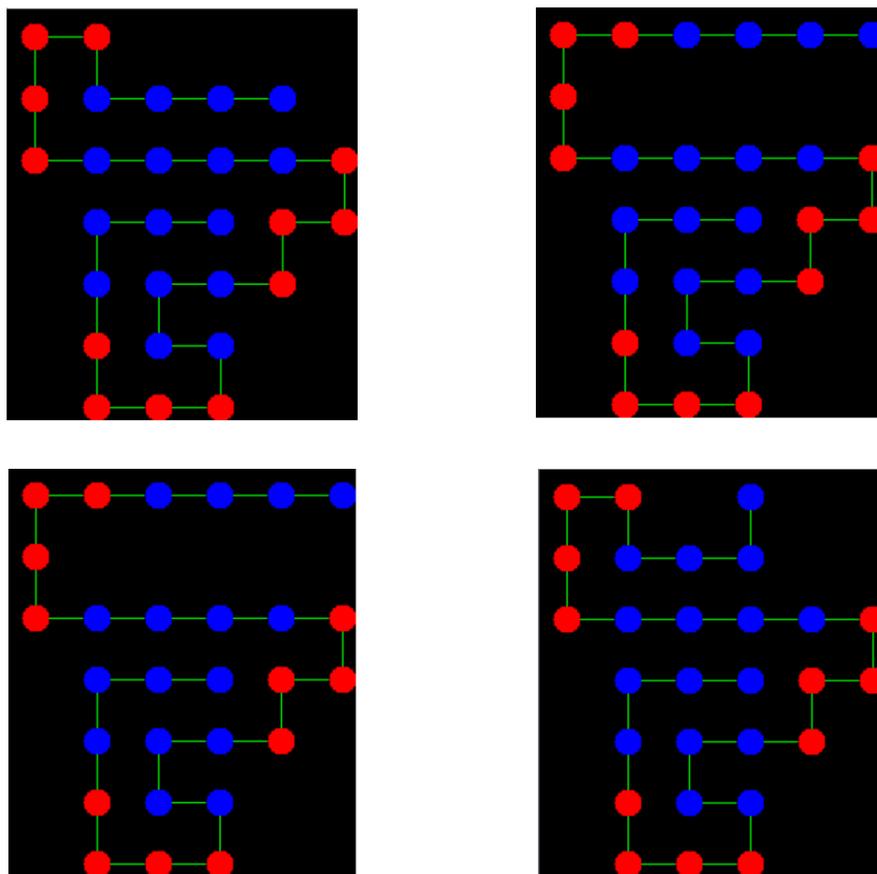
Para proceder a la ejecución de las pruebas, se generaron 200 conformaciones aleatorias 2D cuadrada de la secuencia hidrófoba-polar (HP)  $H_4P_4H_4P_4H_4P_4H_4$ , integrada por 28 elementos. La Figura 10.5 ilustra algunos ejemplos de las 200 conformaciones aleatorias 2D generadas.

**Tabla 10.2** Componentes, clases y métodos a probar durante las pruebas de componentes (pruebas unitarias) de los operadores del algoritmo genético en la plataforma bioinformática Evolution.

Componente	Descripción	Clases del componente	Métodos de la clase
<b>Operator.Selection</b>	Operador de Selección del algoritmo genético.	PopulDecim.java	public boolean population ()
		Roulette.java	public boolean roulette ()
		TopPercent.java	public boolean percent ()
		Tournament.java	public boolean tournament ()
		Selection.java	-
<b>Operator.Crossover</b>	Operador de Cruce del algoritmo genético.	CrossOver.java	public int[] onePointCrossOver () public int[] twoPointCrossOver () public int[] uniformCrossOver ()
<b>Operator.Mutation</b>	Operador de mutación del algoritmo genético.	RankGAmutation.java	public void mutar ()
		RankGAcaterpillar Mutation.java	public void mutar ()
		RankGAclamp Mutation.java	public void mutar ()
		Mutation.java	-

Posteriormente, para cada una de las 200 conformaciones se obtuvo su correspondiente vector de posiciones en el espacio 2D, escrito en el alfabeto {1, 2, 3}, donde: 1 significa movimiento hacia adelante, 2 significa movimiento hacia la derecha, y 3 significa movimiento hacia la izquierda. Algunos ejemplos de los vectores de posiciones obtenidos son ilustrados en la Tabla 10.3.

A partir del conjunto de vectores de posiciones en el espacio 2D generado, fue posible ejecutar las pruebas unitarias de los componentes, clases y métodos que integran el algoritmo evolutivo, relacionados en la Tabla 10.2. Es decir, un conjunto de 200 vectores de posiciones, correspondientes a las 200 conformaciones generadas de forma aleatoria, constituyó el insumo de entrada para las pruebas unitarias de los diferentes tipos de operadores de selección, cruce y mutación (ver Tabla 10.2). Los resultados de la primera iteración de las pruebas unitarias se pueden apreciar en la Tabla 10.4.



**Figura 10.5** Ejemplos de conformaciones 2D generadas aleatoriamente a partir de la secuencia HP H<sub>4</sub>P<sub>4</sub>H<sub>4</sub>P<sub>4</sub>H<sub>4</sub>P<sub>4</sub>H<sub>4</sub> para proceder a las pruebas unitarias de los componentes, clases y métodos relacionados en la Tabla 10.2.

**Tabla 10.3** Ejemplos de vectores de posiciones obtenidos a partir de las conformaciones generadas aleatoriamente, como las mostradas en la Figura 10.5.

Secuencia	Vector de posiciones generado aleatoriamente
<b>S<sub>11</sub></b>	{1,1,1,1,2,3,3,1,3,1,1,1,2,2,3,1,1,3,3,2,2,1,1,2,1,1,2,1}
<b>S<sub>78</sub></b>	{1,1,1,1,1,1,3,1,3,1,1,1,1,2,2,3,2,1,3,3,2,2,1,2,1,1,2,1}
<b>S<sub>145</sub></b>	{1,1,1,1,1,1,3,1,3,1,1,1,1,2, 2,3,2,1,3,3,2,2,1,2,1,1,2,1}
<b>S<sub>189</sub></b>	{1,1,2,1,2,3,3,1,3,1,1,1,1,2,2,3,2,1,3,3,2,2,1,2,1,1,2,1}

**Tabla 10.4** Resultados de la primera iteración de las pruebas de componentes (pruebas unitarias) de los operadores del algoritmo genético en la plataforma bioinformática Evolution.

Componente	Clases del componente	Métodos de la clase	Resultado de la ejecución
<b>Selection</b>	PopulDecim.java	public boolean population ()	Exitoso
	Roulette.java	public boolean roulette ()	Exitoso
	TopPercent.java	public boolean percent ()	Exitoso
	Tournament.java	public boolean tournament ()	Es necesario revisar el algoritmo, ya que se repiten demasiado los mejores elementos de la población.
	Selection.java	-	-
<b>Crossover</b>	CrossOver.java	public int[] onePointCrossOver ()	Exitoso
		public int[] twoPointCrossOver ()	Los dos puntos de cruce se generan de forma correcta. Sin embargo, el intercambio de los segmentos de los cromosomas representando a los dos padres es erróneo.
		public int[] uniformCrossOver ()	La máscara de valores binarios (0,1), para la selección de los genes que permitirán la formación de los dos hijos, no es generada correctamente.
<b>Mutation</b>	RankGAMutation.java	public void mutar ()	Exitoso
	RankGAcaterpillarMutation.java	public void mutar ()	Exitoso
	RankGAclampMutation.java	public void mutar ()	Exitoso
	Mutation.java	-	-

Con relación a la Tabla 10.4, aquí es necesario hacer notar lo siguiente:

- 1) Los métodos *population()*, *roulette()*, *percent()* y *tournament()* de las clases *PopulDecim*, *Roulette*, *TopPercent* y *Tournament*, respectivamente, del componente *Selection*, reciben como parámetro de entrada el conjunto de los 200 vectores de posiciones, como los ilustrados en la Tabla 10.3; produciendo como salida un nuevo conjunto de 200 vectores de posiciones,

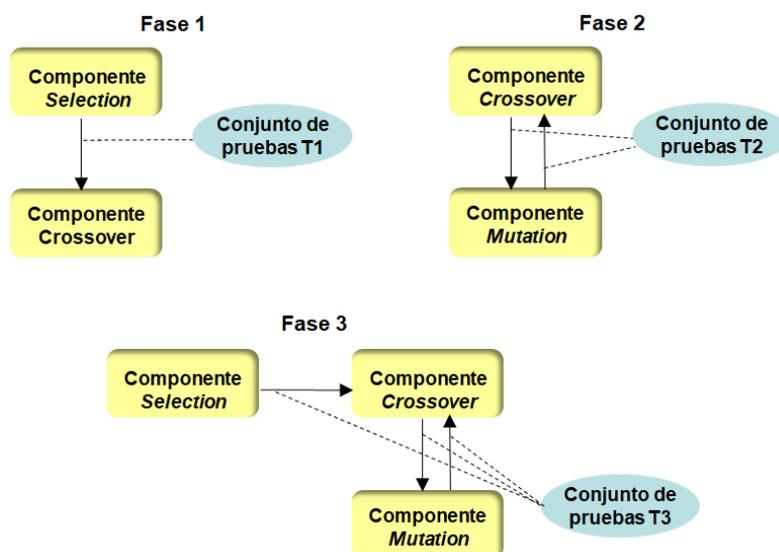
conteniendo los mejores elementos de la población, según la función objetivo evaluada.

- 2) Los métodos *onePointCrossOver()*, *twoPointCrossOver()* y *uniformCrossOver()* de la clase *CrossOver* en el componente *Crossover*, reciben como parámetro dos vectores de posiciones, los cuales fungen como padres o ancestros; produciendo como salida dos nuevos vectores de posiciones (hijos) resultado del cruce de los padres o ancestros.
- 3) El método *mutar()* en las clases *RankGAmutation*, *RankGAcaterpillarMutation* y *RankGAclampMutation* del componente *Mutation*, recibe como parámetro el valor de una posición (gen) de un vector de posiciones, y en dependencia de un valor de probabilidad generado aleatoriamente, producirá como salida el valor de dicha posición mutado o sin mutar.

La fase de pruebas unitarias concluyó cuando la ejecución de cada método de cada clase en cada componente fue la esperada.

#### X.4.2 Pruebas de integración en la plataforma computacional Evolution.

Para ilustrar las pruebas de integración en la plataforma bioinformática Evolution, nos centraremos en la funcionalidad del componente a nivel superior “Algoritmo Evolutivo”, la cual emerge como resultado de la integración de los componentes *Selection*, *Crossover* y *Mutation*, cuyas pruebas unitarias ilustramos en el epígrafe anterior. La Figura 10.6 ilustra el proceso de integración de los tres componentes que definen el comportamiento del “Algoritmo Evolutivo”.



**Figura 10.6** Pruebas de integración del componente a nivel superior “Algoritmo Evolutivo” de la plataforma Bioinformática Evolution.

Como se puede apreciar en la Figura 10.6, la Fase 1 de las pruebas de integración fue dedicada a probar la integración de los componentes *Selection* y *Crossover* del *Algoritmo Evolutivo* (conjunto de pruebas T1). Por otra parte, la Fase 2 comprendió la integración de los componentes *Crossover* y *Mutation* (conjunto de pruebas T2). Finalmente, la Fase 3 abarcó la integración de los tres componentes que conforman el *Algoritmo Evolutivo*, esto es *Selection*, *Crossover* y *Mutation* (conjunto de pruebas T3). Los resultados de las pruebas de integración del componente a nivel superior *Algoritmo Evolutivo*, en su primera iteración, se pueden apreciar en la Tabla 10.5. Aquí es necesario hacer notar que, todas las clases y métodos (ya probados durante las pruebas unitarias), que conforman cada uno de los tres componentes, fueron ejecutados durante las pruebas de integración.

**Tabla 10.5** Resultados de la primera iteración de las pruebas de integración de los tres componentes que integran el Algoritmo Genético en la plataforma bioinformática Evolution.

Componentes integrados	Interacción entre componentes	Resultado de la ejecución
<b>Selection-Crossover</b>	De la población conformada por los mejores individuos (vectores de posiciones) generada por el componente <i>Selection</i> , se selecciona una pareja de vectores de posiciones en forma aleatoria, la cual constituye el valor del parámetro de entrada del componente <i>Crossover</i> . Este último se encarga de generar dos nuevos vectores de posiciones (hijos) a partir de los dos vectores de posiciones (padres) recibidos.	Exitoso
<b>Crossover-Mutation</b>	Durante el proceso de cruce, el componente <i>Crossover</i> invoca el componente <i>Mutation</i> , y le pasa como parámetro el valor de la posición del vector de posiciones (padre) que heredará el nuevo vector de posiciones (hijo). El componente <i>Mutation</i> , a partir de un valor de probabilidad generado aleatoriamente, decide si mutar o no el valor de la posición recibida. Posteriormente, regresa el valor mutado o no mutado al componente <i>Crossover</i> .	Exitoso
<b>Selection-Crossover-Mutation</b>	Se integran en una misma prueba las dos interacciones explicadas previamente.	Exitoso

## X.5 Referencias del capítulo

1. Cusumano, M.A. The Factory Approach to Large-Scale Software Development: Implications for Strategy, Technology, and Structure. Classic Reprints Series, 2017.
2. Garland, J., Anthony, R. Large Scale Software Architecture. A Practical Guide Using UML. Wiley, 2003.
3. Janakiram, D. Building Large Scale Software Systems. McGraw Hill Education, 2013.
4. Lakos, J. Large-Scale C++ Software Design. Addison Wesley, 1996.
5. Screerer, A. Coordination in Large-Scale Agile Software Development: Integrating Conditions and Configurations in Multiteam Systems (Progress in IS). Springer, 2017.
6. Pfleeger, S. L. Ingeniería de software: Teoría y práctica. Pearson Education, 2002.
7. Pressman, R. S. Ingeniería del software: Un enfoque práctico. McGraw-Hill, 2010.
8. Sommerville, I. Ingeniería del software. Pearson Addison Wesley, 2012.
9. Tsui, F., Karam, O., Bernal, B. Essentials of software engineering. Jonas & Bartlett Learning books, 2015.
10. Van Vliet, H. Software engineering: Principles and practice. John Wiley & Sons, 2000.
11. Whittaker, J. A. *How to break software: A practical guide to testing*. Addison-Wesley. England, 2002.

# Capítulo XI Métricas del Proceso de Desarrollo de Software y del Software

## XI.1 Principios de medición

La planeación y estimación son aspectos cruciales dentro de la gestión de un proyecto de software a gran escala [1-6]. La medición del software se refiere a la obtención de un valor numérico, a partir de algún atributo del proceso de desarrollo o del producto software, tales como [7-9]:

- Tamaño del software en Líneas De Código (LDC).
- Funcionalidad del software en Puntos de Función (PF), Objetos, Componentes, etc.
- Planeación (tiempo en meses).
- Esfuerzo (personas-mes o individual).
- Calidad.
- Productividad.
- Fiabilidad, seguridad, ejecución, etc.

A partir de estos valores numéricos y de los estándares observados, es posible arribar a conclusiones acerca tanto del proceso de desarrollo del software como de la calidad del producto software.

## XI.2 Métricas del proceso de desarrollo y métricas del software

¿Por qué medir el proceso de desarrollo y el producto software?

- Para estimar y hacer predicciones acerca de importantes parámetros del proceso de desarrollo tales como planeación (tiempo en meses), esfuerzo (personas-mes o individual), etc.
- Para establecer metas a futuro que aseguren la calidad del producto software.

En la Tabla 11.1 se relacionan las métricas comúnmente utilizadas para estimar y hacer predicciones tanto del proceso de desarrollo como del producto software [7-9].

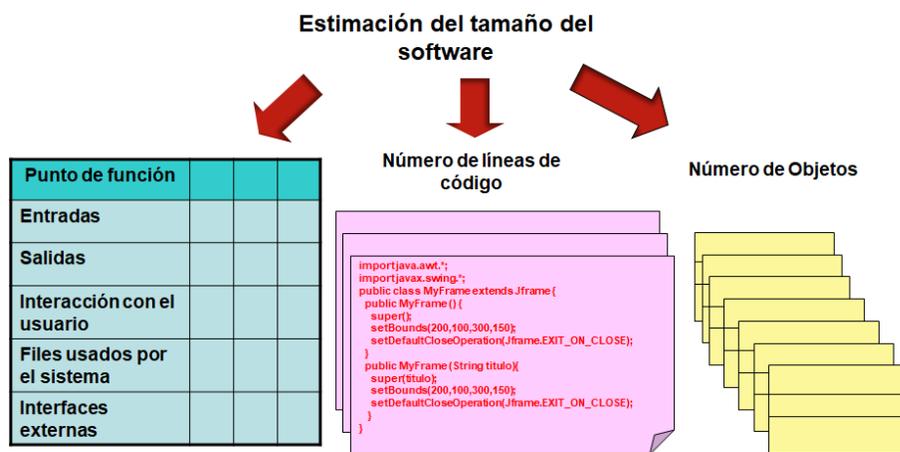
**Tabla 11.1** Métricas del Proceso de Desarrollo y Métricas del Producto Software.

Métrica	Descripción
<b>Métricas orientadas al tamaño del software</b>	Son medidas directas del producto software (en cuanto a su tamaño) y de la calidad del proceso de desarrollo del software. El tamaño del software comúnmente se estima en líneas de código (LDC) y en miles de líneas de código (KLDC). Una vez conocida la estimación del tamaño del software, entonces es posible estimar el esfuerzo personas-mes y la planeación en meses.
<b>Métricas orientadas a la función del software (son también una medida del tamaño del software)</b>	Son medidas indirectas del producto software y de la calidad del proceso de desarrollo. Este tipo de métricas no utiliza las LDC o KLCD, sino que se centra en la funcionalidad del software, utilizando un método conocido como Puntos de Función (PF).
<b>Métricas orientadas a las personas</b>	Son medidas sobre cómo los miembros del equipo desarrollan software, así como sobre la efectividad de los métodos y herramientas.
<b>Métricas orientadas a la calidad</b>	Son medidas de como el software desarrollado satisface los requerimientos, necesidades y expectativas del cliente. La calidad viene calculada a partir del número de errores cometidos y del tamaño del software en KLCD o PF.

<p><b>Métricas orientadas a la productividad</b></p>	<p>Este tipo de métrica se basa en el rendimiento del proceso de desarrollo de software. La productividad viene calculada a partir del tamaño del software en KLCD o PF y el parámetro persona-mes.</p>
<p><b>Métricas orientadas a la técnica</b></p>	<p>Son medidas que se centran en características del software - tales como grado de modularidad, complejidad de la arquitectura, etc. - más que en el proceso de desarrollo del software.</p>

Una estimación efectiva necesita estimar primero el tamaño del software antes de estimar la planeación. Como se ilustra en la Figura 11.1, existen varios enfoques para estimar el tamaño del software [6-9], siendo comúnmente utilizados los siguientes:

- Enfoque algorítmico, como los puntos de función, para estimar el tamaño del programa a partir de los requerimientos funcionales, no funcionales, así como de factores que influyen en la complejidad del sistema a desarrollar.
- Estimación del tamaño del programa, a partir de la descripción de los macro requerimientos del programa (interfaces interactivas, funcionalidad lógica, persistencia de datos, etc.).
- Si ya se ha trabajado en proyectos similares y se conoce el tamaño de éstos en líneas de código, entonces es posible estimar cada una de las partes principales del nuevo sistema como un porcentaje del tamaño de una parte similar del sistema conocido. Estimar el tamaño total del nuevo sistema sumando los tamaños estimados de cada una de las partes. Este enfoque resulta ser muy útil cuando se cuenta con información valiosa sobre sistemas previamente desarrollados.



**Figura 11.1** Principales enfoques para la estimación del tamaño del software.

La estimación de un proyecto software será buena en la medida en que sea buena la estimación del tamaño del producto. De aquí que la estimación del tamaño represente la primera tarea importante del planeador de proyectos. En el contexto de la planeación de proyectos de software, el tamaño se refiere a una producción cuantificable del software.

Las estimaciones de Líneas de Código (LDC) y Puntos de Función (PF) son técnicas de estimación diferentes, a pesar de que ambas tienen varias características en común. El planeador del proyecto de software comienza con un enfoque limitado para el ámbito del software (declaración inicial del alcance del software) y desde esta declaración intenta descomponer el software en funcionalidades que se pueden estimar individualmente. Para cada función entonces se estiman las LDC y/o los PF.

Las técnicas de estimación de LDC y PF difieren en el nivel de detalle que se requiere para la descomposición y el objetivo de la partición. Cuando se utiliza LDC como variable de estimación, la descomposición es absolutamente esencial (en general se descomponen las funcionalidades del sistema, aunque también se puede utilizar una lista de componentes estándar).

Para estimaciones de PF, la descomposición funciona de diferente manera. En lugar de centrarse en las funcionalidades, se estiman cada una de las características del dominio de la información [6-9] - entradas, salidas, archivos de datos, peticiones, e interfaces externas - y catorce valores de ajuste de la complejidad (como veremos más adelante). Las estimaciones resultantes se utilizan para derivar un valor de PF que se pueda utilizar para generar una estimación.

Independientemente de la variable de estimación que se utilice (LDC o PF), el planeador del proyecto comienza por estimar un rango de valores para cada función o valor del dominio de la información. Mediante los datos históricos o la intuición (esto último cuando todo lo demás falla), el planeador estima un valor de tamaño “optimista”, “más probable”, y “pesimista” para cada función [6-9]. Al especificar un rango de valores, se proporciona una indicación implícita del grado de incertidumbre.

Posteriormente, se calcula un valor de tres puntos o esperado. El valor esperado de la variable de estimación (tamaño) VE, se puede calcular como una media ponderada de las estimaciones optimistas ( $S_{opt}$ ), las más probables ( $S_m$ ), y las pesimistas ( $S_{pess}$ ):

$$VE = (S_{opt} + 4S_m + S_{pess})/6$$

### XI.3 Estimación de los puntos de función

Un punto de función (PF) [7, 8, 11-13] es una medida sintética del tamaño de un programa, que se suele utilizar en los primeros estados del proyecto. Un punto de función no es una funcionalidad particular de un sistema como, por ejemplo, “transferencia interbancaria” (en un portal bancario) o “reservación de vuelo” (en el

sitio Web de una aerolínea). Los puntos de función se refieren a las características del dominio de información y son más fáciles de determinar a partir de la especificación de los requerimientos que las líneas de código (LDC), proporcionando una medida más exacta sobre el tamaño del programa.

Según el método IBM [7, 8, 11-13], el número de puntos de función en un programa se basa en el número y complejidad de cada uno de los elementos siguientes:

- **Entradas.** Interfaces gráficas de usuario, formularios, cuadros de diálogo, controles o mensajes, a través de los cuales un usuario final o cualquier otro programa pueda interactuar con el sistema de software.
- **Salidas.** Interfaces gráficas de usuario, cuadros de texto, gráficos o mensajes que el programa genera para el usuario final o cualquier otro programa.
- **Consultas.** Combinaciones de entrada/salida o peticiones en las que cada entrada genera una salida inmediata.
- **Archivos lógicos internos.** Son los principales grupos lógicos de datos de usuarios finales o información de control que están completamente controlados por el programa. Un archivo lógico podría estar integrado de un único archivo plano o de una sola tabla en una base de datos relacional, orientada a objetos, etc.
- **Archivos de interfaz externos.** Archivos controlados por otros programas, con los que el programa va a interactuar (bases de datos externas, grupos de datos lógicos o información de control que entre o salga del programa).

Para calcular el número de puntos de función en un programa se siguen los pasos que se relacionan a continuación [7, 8, 11-13]:

- 1.- Calcular el **total de los puntos de función sin ajustar (PF)**, utilizando los multiplicadores de puntos de función que se presentan en la Tabla 11.2. Es decir, el número resultante de cada una de las características del software (por ejemplo, número de consultas de usuario), dependiendo de la complejidad asignada, debe multiplicarse por el número indicado en la correspondiente columna.

**Tabla 11.2** Multiplicadores para el cálculo de los puntos de función sin ajustar.

Características del software	Complejidad baja	Complejidad media	Complejidad alta
Número de entradas de usuario	x 3	x 4	x 6
Número de salidas de usuario	x 4	x 5	x 7
Número de consultas de usuario	x 3	x 4	x 6
Número de archivos	x 7	x 10	x 15
Archivos de interfaz externos	x 5	x 7	x 10
Subtotal de Puntos de Función sin Ajustar (PF)			
Total de Puntos de Función sin Ajustar (PF)			

2.- Calcular el **multiplicador de influencia**. Este **multiplicador de influencia** está basado en la influencia que tienen **14 factores** sobre el desarrollo de un proyecto de software. Estos factores incluyen **comunicaciones de datos, entrada de datos en línea, complejidad del procesamiento, y facilidad de instalación**, entre otros. El intervalo del multiplicador de influencia es **[0.65, 1.35]**. La **escala de “valores de ajuste de la complejidad”** se muestran en la Figura 11.2 y los **14 factores** que deben ser evaluados se en la Tabla 11.3.



Figura 11.2 Escala de valores de ajuste de la complejidad.

**Tabla 11.3** Factores que influyen sobre el desarrollo de un proyecto de software [7-9].

<b>Factores</b>
1. ¿Requiere el sistema copias de seguridad y de recuperación fiables?
2. ¿Se requiere comunicación de datos?
3. ¿Existen funciones de procesamiento distribuido?
4. ¿Es crítico el rendimiento?
5. ¿Se ejecutará el sistema en un entorno operativo existente y fuertemente utilizado?
6. ¿Requiere el sistema entrada de datos interactiva?
7. ¿Requiere la entrada de datos interactiva que las transacciones de entrada se lleven a cabo sobre múltiples pantallas u operaciones?
8. ¿Se actualizan los archivos maestros de forma interactiva?
9. ¿Son complejas las entradas, las salidas, los archivos o las peticiones?
10. ¿Es complejo el procesamiento interno?
11. ¿Se ha diseñado el código para ser reutilizable?
12. ¿Están incluidas en el diseño la conversión y la instalación?
13. ¿Se ha diseñado el sistema para soportar múltiples instalaciones en diferentes organizaciones?
14. ¿Se ha diseñado la aplicación para facilitar los cambios y para ser fácilmente utilizada por el usuario?

3.- Calcular el **total de puntos de función ajustados (PFA)**, utilizando el **multiplicador de influencia** previamente calculado. Para el cálculo de los puntos de función ajustados (PFA) se utiliza la siguiente expresión:

$$PFA = PF * \left[ 0.65 + 0.01 \sum_i f_i \right]$$

donde: **PF** es el total de puntos de función sin ajustar, y  $\sum f_i$  es la suma de los valores de ajuste de la complejidad. La tabla previamente construida (ver Tabla 11.2) quedará actualizada como se muestra en la Tabla 11.4.

**Tabla 11.4** Multiplicadores para el cálculo de los puntos de función sin ajustar, total de puntos de función sin ajustar y total de puntos de función ajustados [7-9].

<b>Características del software</b>	<b>Complejidad baja</b>	<b>Complejidad media</b>	<b>Complejidad alta</b>
Número de entradas de usuario	x 3	x 4	x 6
Número de salidas de usuario	x 4	x 5	x 7
Número de consultas de usuario	x 3	x 4	x 6

Número de archivos	x 7	x 10	x 15
Archivos de interfaz externos	x 5	x 7	x 10
Subtotal de Puntos de Función sin Ajustar (PF)			
Total de Puntos de Función sin Ajustar (PF)			
Multiplicador			
Total de Puntos de Función Ajustados (PFA)			

Al final del capítulo, se ilustrará de forma íntegra la estimación del número de puntos de función, a través del caso de estudio Evolution: Una Plataforma Computacional para el Modelado y Simulación del Plegamiento de Proteínas.

## XI.4 Estimación del Tamaño del Software en Líneas de Código a Partir de los Puntos de Función

La relación entre **Líneas de Código (LDC)** y **Puntos de Función (PF)** depende del **Lenguaje de Programación** que se utilice para implementar el software. Para algunos lenguajes en particular hay una gran correlación entre **LDC** y **PF**. La estimación del número de **LDC** a partir del número de **PF** es poco exacta e informal, ya que existen otros muchos factores que podrían afectar dicha conversión. Sin embargo, tener a la mano un grupo de estimaciones, aún informales, de dicha conversión a los principales lenguajes de programación puede resultar de gran valor a la hora de estimar el **esfuerzo personas-mes** y la **planeación en meses**, ya que la gran mayoría de los algoritmos y estrategias para estimar estas dos importantes medidas se basan en el **tamaño del software** en **LDC**.

No obstante, a la gran gama de recursos (bibliotecas de código, principios de diseño e implementación basada en la reusabilidad, disponibilidad de componentes reutilizables, entre otros), desde hace décadas continúa siendo ampliamente aceptado en las estimaciones del software que un **PF** equivale aproximadamente a 50-60 **LDC** en un lenguaje orientado a objetos, tales como C++, Java, entre otros. La Tabla 11.5 proporciona el número estimado de LDC correspondiente a un PF para varios de los lenguajes orientados a objetos comúnmente utilizados.

**Tabla 11.5** Número estimado de LDC correspondientes a un PF en lenguajes de programación orientados a objetos.

Lenguaje de programación orientada a objetos	Número de LDC promedio por PF
<b>C++</b>	59
<b>C#</b>	58
<b>Java</b>	55
<b>JavaScript</b>	54
<b>JSP</b>	59
<b>.NET</b>	60
<b>PHP basado en framework</b>	15 - 20

## XI.5 Estimación del esfuerzo personas-mes y de la planeación en meses

Una vez que se tiene la estimación del tamaño del software en LDC o PF, entonces se puede derivar la estimación del esfuerzo personas-mes y de la planeación en meses.

La estimación del esfuerzo es necesaria para poder saber a cuántas personas hay que incorporar en el proyecto; además, teniendo una estimación del esfuerzo se facilita la estimación de la planeación [7, 8, 11-13]. Derivar la estimación del esfuerzo es un proceso sencillo y comúnmente ésta se expresa en personas-mes. Por otra parte, la estimación de la planeación en meses es requerida para conocer el tiempo aproximado que conllevará el desarrollo del sistema de software, así como para planear la secuencia y tiempo que se dedicará a cada una de las actividades que forman parte del proceso de desarrollo de software.

La estimación del esfuerzo personas-mes y de la planeación en meses puede ser obtenida directamente a partir de la estimación del tamaño del software, tanto en LDC como en PF. Algunas de las alternativas para lograr esto son las siguientes:

- Utilizar un software de estimación para crear directamente la estimación del esfuerzo personas-mes y de la planeación en meses a partir de la estimación del tamaño en LDC o en PF.
- Utilización de tablas de planeación [7] para convertir la estimación del tamaño en LDC a una estimación del esfuerzo personas-mes y de la planeación en meses.
- Utilización de datos anteriores de la organización acerca de la estimación del esfuerzo en proyectos anteriores similares.
- Utilización de un método algorítmico de aproximación para convertir la estimación de las líneas de código en estimación del esfuerzo. Por ejemplo,

el modelo COCOMO de Barry Boehm o el modelo del ciclo de vida de Putnam y Myers [7, 8].

### XI.5.1 Estimación del esfuerzo personas-mes y de la planeación en meses a partir de las tablas de planeación

Las tablas de planeación establecen una relación entre tamaño del sistema (en líneas de código), esfuerzo personas-mes y planeación en meses. Esta relación es particular para cada uno de los principales tipos de productos existentes: software de sistemas, software de gestión, y productos “*pret-à-porter*” (listo para llevar) [7, 8]. Comúnmente, el tamaño del software en las tablas de planeación viene expresado en LDC, ya que si no se conoce de antemano el lenguaje de programación sería muy complicado derivar el esfuerzo a partir de los PF. La estructura comúnmente usada en una tabla de planeación se ilustra en la Tabla 11.6, en la cual los valores registrados a modo de ilustración fueron tomados de la Tabla de Planeaciones Nominales publicada en [7].

**Tabla 11.6** Estructura de una tabla de planeación para la estimación del esfuerzo personas-mes y de la planeación en meses [7].

Tamaño del sistema en LDC	Productos de Sistemas		Productos de Gestión		Productos “ <i>prêt-à-porter</i> ”	
	Planeación (meses)	Esfuerzo (personas-mes)	Planeación (meses)	Esfuerzo (personas-mes)	Planeación (meses)	Esfuerzo (personas-mes)
10,000	10	48	6	9	7	15
25,000	15	140	9	27	10	44
40,000	18	270	10	54	13	88
80,000	24	630	14	125	17	210

Como se puede apreciar en la Tabla 11.6, para estimar el esfuerzo personas-mes utilizando la tabla de planeación [7], se debe haber estimado previamente el tamaño del software en LDC y clasificar el sistema de software a desarrollar en una de las siguientes tres categorías: software de sistema, software de gestión o software a la medida (*prêt-à-porter*).

### XI.5.2 Estimación de la planeación en meses a través del método de estimación de primer orden de Jones

Si se tiene la suma total de todos los PF, entonces se puede realizar a partir de ellos un cálculo aproximado de la planeación, utilizando el método de Jones [7-9]. Para utilizar este método, simplemente hay que tomar el total de PF y elevarlo a la potencia apropiada, seleccionándola en la Tabla 11.7, tal y como se muestra en la siguiente expresión:

$$\text{Planeación} = (PF)^{exp}$$

Los exponentes de la Tabla 11.7 fueron derivados de un análisis desarrollado por Jones a partir de una base de datos de miles de proyectos [7-8].

**Tabla 11.7** Método de Jones. Exponentes para calcular la planeación en meses a partir de los PF [7-8].

Clase de software	Mejor caso	Media	Peor caso
Sistemas	0.43	0.45	0.48
Gestión	0.41	0.43	0.46
Prêt-à-porter	0.39	0.42	0.45

### XI.5.3 El modelo COCOMO para la estimación del esfuerzo personas-mes y la planeación en meses

La familia de modelos **COCOMO (Constructive Cost Model)** constituyen otra valiosa alternativa para la estimación del **esfuerzo personas-mes** y de la **planeación en meses** a partir del **tamaño del software en miles de líneas de código (KLDC)** y otros aspectos relacionados con los costos. Aquí solo nos referiremos a las expresiones para la estimación del **esfuerzo personas-mes** y de la **planeación en meses**, según el modelo **COCOMO Básico**, el cual únicamente se basa en el **tamaño del software en KLDC**. Para una mayor comprensión y detalles del alcance de la familia de modelos COCOMO consultar [7-9 y 12]. Según el modelo COCOMO Básico, el esfuerzo personas-mes (E) y la planeación en meses (D) se estiman a partir de las siguientes expresiones:

$$E = a_b KLDC^{b_b}$$

donde: KLDC es el número de miles de líneas de código,  $a_b$  y  $b_b$  son el coeficiente y exponente, respectivamente (ver Tabla 11.8) y E es el esfuerzo estimado.

$$D = c_b E^{d_b}$$

donde: E es el Esfuerzo personas-mes estimado,  $c_b$  y  $d_b$  son el coeficiente y exponente, respectivamente (ver Tabla 11.8) y D es la planeación en meses estimada.

**Tabla 11.8** Coeficientes y exponentes para el modelo COCOMO Básico [7-8].

Tipo de proyecto de software	$a_b$	$b_b$	$c_b$	$d_b$
Orgánico (pequeños en tamaño y complejidad)	2.4	1.05	2.5	0.38
Semiacoplado (intermedio en tamaño y complejidad)	3.0	1.12	2.5	0.35
Empotrado (desarrollados para un tipo de hardware particular con fuertes restricciones operativas)	3.6	1.20	2.5	0.32

## XI.6 Métricas estáticas y métricas dinámicas

Las **métricas estáticas del software** [9] se refieren a aquellas mediciones efectuadas durante el proceso de desarrollo del software, a través de las etapas de análisis de requerimientos, diseño, codificación, etc. Ejemplos de métricas estáticas del software son las métricas orientadas al tamaño del software (KLDC) y las métricas orientadas a la función del software (PF). Por otra parte, las **métricas dinámicas del software** [9] se refieren a las mediciones efectuadas durante la ejecución del sistema o programa.

Como se puede apreciar en la Figura 11.3, las métricas estáticas del software son útiles para valorar la comprensión, complejidad y mantenimiento del sistema de software; por su parte, las métricas dinámicas están relacionadas con los atributos de calidad del software tales como: fiabilidad, confiabilidad, eficiencia, ejecución, portabilidad, seguridad, etc.

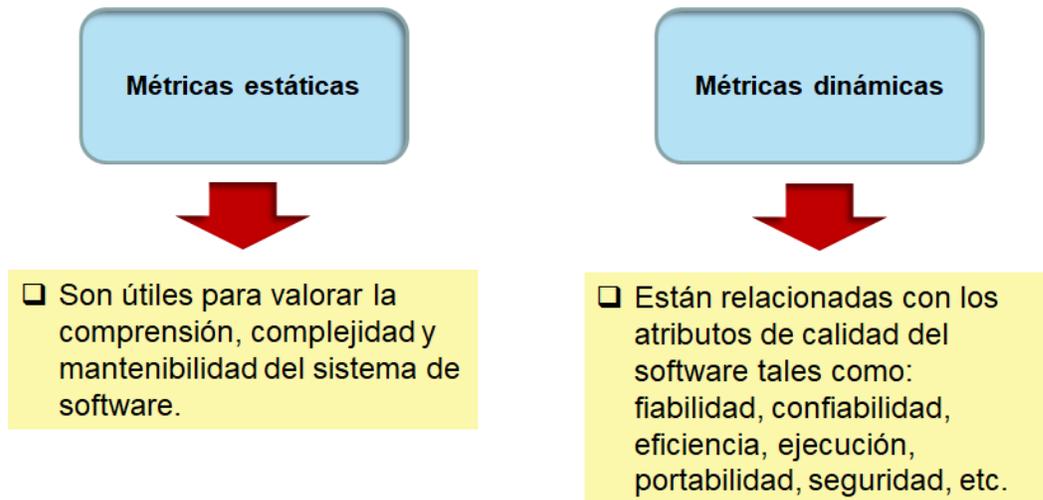


Figura 11.3 Métricas estáticas y métricas dinámicas del software.

En el caso particular del software orientado a objetos, los siguientes son ejemplos de métricas estáticas [9]:

- Profundidad del árbol de herencia.
- Fan-in/Fan-out: Medida del número de métodos que llaman a otro método X / Medida del número de métodos que son llamados por un método X.
- Métodos pesados por clase: Métodos incluidos en una clase con sus correspondientes pesos, los cuales vienen dados por la complejidad del método (valor de 1 para un método sencillo, y valores más grandes para métodos más complejos).
- Número de operaciones sobrescritas: número de operaciones de una superclase que son sobrescritas en la subclase.

## XI.7 Caso de Estudio

### XI.7.1 Plataforma computacional Evolution: un laboratorio virtual para el modelado y simulación del plegamiento de proteínas

#### XI.7.1.1 Estimación del tamaño del software en puntos de función

#### Paso 1: Recolección inicial de los requerimientos

##### Requerimientos funcionales:

1. Generación y visualización de una población aleatoria de conformaciones 2D Cuadrada a partir de una secuencia HP proporcionada como entrada.
2. Interacción con las conformaciones 2D Triangular.
3. Generación y visualización de una población aleatoria de conformaciones 2D Triangular a partir de una secuencia HP proporcionada como entrada.
4. Interacción con las conformaciones 2D Triangular.
5. Generación y visualización de una población aleatoria de conformaciones 3D Cúbica a partir de una secuencia HP proporcionada como entrada.
6. Interacción con las conformaciones 3D Cúbica.
7. Generación y visualización de los gráficos 2D (fitness vs. conformación, radio de giro vs. Conformación, etc.) para describir las características del espacio de conformaciones generado.
8. Optimización de la población de conformaciones generada a través del uso de un algoritmo genético que integre una amplia gama de operadores genéticos y técnicas genéticas.
9. Generación y visualización de los gráficos 2D (fitness vs. generación, radio de giro promedio vs. generación, distancia máxima vs. generación, relación fitness padres/hijo, etc.) para describir las características del proceso de optimización a través de las diferentes generaciones creadas.
10. Mecanismo de migración genética que permita retomar una generación particular creada por el algoritmo genético y sustituir algunas de sus conformaciones por nuevo material genético creado de forma aleatoria.

##### Requerimientos no funcionales:

1. Sistema fuertemente visual, dinámico e interactivo que basa todo su comportamiento en la interacción del usuario con componentes activos que integran diferentes interfaces gráficas.

2. Características de un Laboratorio Virtual donde los experimentos pueden iniciarse, detenerse, almacenarse, recuperarse, modificarse, continuarse, etc.
3. Arquitectura monolítica de tres capas que garantice un fuerte desacoplamiento entre la representación de los datos, presentación de los datos y lógica de la aplicación, que garantice una fácil migración de la aplicación a una arquitectura cliente-servidor web.
4. Eficiencia y eficacia en la ejecución de los algoritmos de plegamiento, así como en la administración de la memoria RAM requerida.
5. Reusabilidad y extensibilidad del sistema.
6. Portabilidad del sistema.

## **Paso 2: Clasificación de la funcionalidad esperada según las Características del Dominio de la Información**

<b>Interfaces de Entrada</b>
<b>Interfaz Principal con Menú Principal (Project, Run, Tools y About) y los correspondientes Submenús (CA)</b>
<b>Interfaz para introducir la Secuencia de Aminoácidos o HP (CB)</b>
<b>Interfaz para seleccionar el Modelo HP y el Algoritmo de Optimización a ejecutar (CB)</b>
<b>Interfaz para la selección de los Operadores del Algoritmo de Optimización (CM)</b>
<b>Interfaz para introducir el Número de Generaciones a producir (CB)</b>
<b>Interfaz para definir el sampling (CB)</b>
<b>Interfaz para definir el número de experimentos a ejecutar (CB)</b>
<b>Interfaz Principal de Visualización con Menú Principal (Generation Charts, Evolution Charts, Experiments) y Botones Conformational Space, Algorithm Workspace y Genetic Immigration (CA)</b>
<b>Interfaz para la Selección de Generaciones y Conformaciones en la Inmigración Genética (CA)</b>

CA: Complejidad Alta, CM: Complejidad Media, CB: Complejidad Baja

<b>Interfaces de Salida</b>
<b>Interfaz para la Visualización del Espacio de Conformaciones 2D Cuadrado (CA)</b>
<b>Interfaz para la Visualización del Espacio de Conformaciones 2D Triangular (CA)</b>
<b>Interfaz para la Visualización del Espacio de Conformaciones 3D Cúbica (CA)</b>
<b>Interfaz para la Visualización del Algorithm Workspace (CA)</b>
<b>Interfaz para la Visualización del Gráfico Fitness Evolution (CB)</b>
<b>Interfaz para la Visualización del Gráfico Average Radius of Gyration – Average Maximum Diameter of the Conformation (CB)</b>
<b>Interfaz para la Visualización del Gráfico de los Experimentos (CB)</b>
<b>Interfaz para la Visualización del Gráfico Fitness/Conformation para una Generación dada (CB)</b>
<b>Interfaz para la Visualización del Gráfico Average Radius of Gyration – Average Maximum Diameter of the Conformation para una Conformación dada (CB)</b>

---

**Interfaz para la Visualización del Gráfico Parents – OffSprings Fitness para una Conformación dada (CB)**

---

CA: Complejidad Alta, CM: Complejidad Media, CB: Complejidad Baja

---

**Peticiones/Consultas**

---

**Generación del Espacio de Conformaciones 2D Cuadrada (CA)**

---

**Generación del Espacio de Conformaciones 2D Triangular (CA)**

---

**Generación del Espacio de Conformaciones 3D Cúbico (CA)**

---

**Visualización y Manipulación del Espacio de Conformaciones 2D Cuadrada (CA)**

---

**Visualización y Manipulación del Espacio de Conformaciones 2D Triangular (CA)**

---

**Visualización y Manipulación del Espacio de Conformaciones 3D Cúbica (CA)**

---

**Ejecución del Algoritmo Genético (CB)**

---

**Cálculo del Fitness de las Conformaciones (CA)**

---

**Ejecución del Operador de Selección (CA)**

---

**Formación de Parejas (CM)**

---

**Ejecución del Operador de Cruce (CA)**

---

**Ejecución del Operador de Elitismo (CM)**

---

**Visualización del Algorithm Workspace (CA)**

---

**Inmigración Genética (CA)**

---

**Generación y Visualización del Gráfico Fitness Evolution (CM)**

---

**Generación y Visualización del Gráfico Average Radius of Gyration – Average Maximum Diameter of the Conformation (CA)**

---

**Generación y Visualización del Gráfico de los Experimentos (CM)**

---

**Generación y Visualización del Gráfico Fitness/Conformation para una Generación dada (CM)**

---

**Generación y Visualización del Gráfico Average Radius of Gyration – Average Maximum Diameter of the Conformation para una Conformación dada (CM)**

---

**Generación y Visualización del Gráfico Parents – OffSprings Fitness para una Conformación dada (CM)**

---

CA: Complejidad Alta, CM: Complejidad Media, CB: Complejidad Baja

---

**Archivos internos**


---

**Archivo para la persistencia del Blackboard con las Generaciones, Conformaciones y atributos de las mismas (CA)**


---

CA: Complejidad Alta, CM: Complejidad Media, CB: Complejidad Baja

**Interfaces externas**

No se requieren

**Paso 3: Cálculo del total de los puntos de función sin ajustar (PF)**

Características del Software	Complejidad baja	Complejidad media	Complejidad alta
Número de entradas de usuario	$5 \cdot 3 = 15$	$1 \cdot 4 = 4$	$3 \cdot 6 = 18$
Número de salidas de usuario	$6 \cdot 4 = 24$	$0 \cdot 5 = 0$	$4 \cdot 7 = 28$
Número de consultas de usuario	$1 \cdot 3 = 3$	$7 \cdot 4 = 28$	$12 \cdot 6 = 72$
Número de archivos	$0 \cdot 7 = 0$	$0 \cdot 10 = 0$	$1 \cdot 15 = 15$
Número de interfaces externas	$0 \cdot 5 = 0$	$0 \cdot 7 = 0$	$0 \cdot 10 = 0$
<b>Subtotal de Puntos de Función sin Ajustar (PF)</b>	<b>42</b>	<b>32</b>	<b>133</b>
<b>Total de Puntos de Función sin Ajustar (PF)</b>	<b>207</b>		

**Paso 4: Cálculo del multiplicador de influencia**

Factores que influyen sobre el desarrollo de un proyecto de software	Valor de ajuste de la complejidad
1. ¿Requiere el sistema copias de seguridad y de recuperación fiables?	0
2. ¿Se requiere comunicación de datos?	0
3. ¿Existen funciones de procesamiento distribuido?	3
4. ¿Es crítico el rendimiento?	5
5. ¿Se ejecutará el sistema en un entorno operativo existente y fuertemente utilizado?	0
6. ¿Requiere el sistema entrada de datos interactiva?	5
7. ¿Requiere la entrada de datos interactiva que las transacciones de entrada se lleven a cabo sobre múltiples pantallas u operaciones?	5
8. ¿Se actualizan los archivos maestros de forma interactiva?	0
9. ¿Son complejas las entradas, las salidas, los archivos o las peticiones?	4

10. ¿Es complejo el procesamiento interno?	5
11. ¿Se ha diseñado el código para ser reutilizable?	5
12. ¿Están incluidas en el diseño la conversión y la instalación?	4
13. ¿Se ha diseñado el sistema para soportar múltiples instalaciones en diferentes organizaciones?	0
14. ¿Se ha diseñado la aplicación para facilitar los cambios y para ser fácilmente utilizada por el usuario?	5
<b>Suma de los Valores de Ajuste de la Complejidad</b>	<b>41</b>

**Paso 5: Cálculo del total de los puntos de función ajustados (PFA)**

$$PFA = PF * \left[ 0.65 + 0.01 \sum_i f_i \right]$$

$$PF = 207$$

$$\sum_i f_i = 41$$

$$PFA = 207 * [0.65 + 0.01*41]$$

$$PFA = 207 * 1.06$$

$$PFA = 219.42$$

Características del Software	Complejidad baja	Complejidad media	Complejidad alta
Número de entradas de usuario	5*3 = 15	1*4 = 4	3*6 = 18
Número de salidas de usuario	6*4 = 24	0*5 = 0	4*7 = 28
Número de consultas de usuario	1*3 = 3	7*4 = 28	12*6 = 72
Número de archivos	0*7 = 0	0*10 = 0	1*15 = 15
Número de interfaces externas	0*5 = 0	0*7 = 0	0*10 = 0
<b>Subtotal de Puntos de Función sin Ajustar (PF)</b>	<b>42</b>	<b>32</b>	<b>133</b>
<b>Total de Puntos de Función sin Ajustar (PF)</b>	<b>207</b>		
<b>Total de Puntos de Función Ajustados (PFA)</b>	<b>219.42</b>		

### XI.7.2 Cálculo de la planeación utilizando los puntos de función y el método de estimación de primer orden de Jones

Según la Tabla 11.7, la plataforma bioinformática Evolution puede ser clasificada como un software de “Gestión” en la categoría “Media”.

Clase de software	Mejor caso	Media	Peor caso
Sistemas	0.43	0.45	0.48
<b>Gestión</b>	0.41	<b>0.43</b>	0.46
Prêt-à-porter	0.39	0.42	0.45

$$\text{Planeación} = (PF)^{exp}$$

$$PF = 219.42$$

$$exp = 0.43$$

$$\text{Planeación} = (219.42)^{0.43}$$

$$\text{Planeación} = 10.15 \cong 10 \text{ meses}$$

### XI.7.3 Estimación del número de líneas de código a partir del número de puntos de función

Aun siendo poco exacta e informal la estimación del número de **LDC** a partir del número de **PF**, para efectuar la conversión tomaremos en cuenta el siguiente supuesto: “**1 PF equivale aproximadamente a 50-60 LDC en un Lenguaje Orientado a Objetos**” (ver Tabla 11.5). Ahora bien, considerando que aspectos tales como la reusabilidad y extensibilidad del diseño, así como el soporte de la implementación en bibliotecas de código, contribuyen a una significativa reducción del número de líneas de código, asumiremos la siguiente equivalencia:

$$1 PFA = 50 LDC$$

de donde:

$$219.42 PFA = 10,971 LDC \cong 11,000 LDC = 11 KLDC$$

donde: KLDC indica miles de líneas de código.

### XI.7.4 Estimación del Esfuerzo Personas-Mes y de la Planeación en Meses con el método COCOMO

Según la Tabla 11.8, la plataforma bioinformática Evolution puede ser clasificada como un software semiacoplado, intermedio en tamaño y complejidad.

Tipo de proyecto de software	$a_b$	$b_b$	$c_b$	$d_b$
Orgánico (pequeños en tamaño y complejidad)	2.4	1.05	2.5	0.38
<b><u>Semiacoplado (intermedio en tamaño y complejidad)</u></b>	<b><u>3.0</u></b>	<b><u>1.12</u></b>	<b><u>2.5</u></b>	<b><u>0.35</u></b>
Empotrado (desarrollados para un tipo de hardware particular con fuertes restricciones operativas)	3.6	1.20	2.5	0.32

#### Cálculo del Esfuerzo Personas-Mes

$$E = a_b KLDC^{b_b}$$

$$E = 3.0(11)^{1.12}$$

$$E = 3.0(14.66)$$

$$E = 44 \text{ personas} - \text{mes}$$

#### Cálculo de la Planeación en Meses

$$D = c_b E^{d_b}$$

$$D = 2.5(44)^{0.35}$$

$$D = 2.5(3.76)$$

$$D = 2.5(3.76)$$

$$D = 9.4 \text{ meses}$$

### XI.8 Referencias del capítulo

1. Cusumano, M.A. The Factory Approach to Large-Scale Software Development: Implications for Strategy, Technology, and Structure. Classic Reprints Series, 2017.
2. Garland, J., Anthony, R. Large Scale Software Architecture. A Practical Guide Using UML. Wiley, 2003.
3. Janakiram, D. Building Large Scale Software Systems. McGraw Hill Education, 2013.
4. Lakos, J. Large-Scale C++ Software Design. Addison Wesley, 1996.

5. Screerer, A. *Coordination in Large-Scale Agile Software Development: Integrating Conditions and Configurations in Multiteam Systems (Progress in IS)*. Springer, 2017.
6. Pfleeger, S. L. *Ingeniería de software: Teoría y práctica*. Pearson Education, 2002.
7. Pressman, R. S. *Ingeniería del software: un enfoque práctico*. McGraw-Hill, 1998
8. Pressman, R. S. *Ingeniería del software: Un enfoque práctico*. McGraw-Hill, 2010.
9. Sommerville, I. *Ingeniería del software*. Pearson Addison Wesley, 2012.
10. Tsui, F., Karam, O., Bernal, B. *Essentials of software engineering*. Jonas & Bartlett Learning books, 2015.
11. Van Vliet, H. *Software engineering: Principles and practice*. John Wiley & Sons, 2000.
12. Garmus, D., Herron, D. *Function Point Analysis: Measurement Practices for Successful Software Projects*. Addison-Wesley. 2000.
13. Jones, C. *Software Assessments Benchamarks, and Best Practices*. Addison-Wesley. 2000.
14. DeMarco, T. *Controlling Software Projects*. Prentice Hall. 1982.

# Capítulo XII Gestión del Mantenimiento

El mantenimiento de un sistema de software a gran escala [1-5] puede convertirse en un proceso complicado y difícil, si durante las diferentes etapas de desarrollo del mismo no se ha atendido el impacto de las características que exhibe este tipo de sistema, las cuales ya fueron discutidas en el Capítulo I. De aquí que, aspectos tales como modularización, reusabilidad, extensibilidad, entre otros aspectos, sean claves durante el desarrollo de un sistema de software a gran escala, de forma tal que el mantenimiento no sea un proceso complicado y difícil.

## XII.1 Tipos de mantenimiento

El mantenimiento del software [6-10] es una fase posterior al desarrollo y liberación del producto software y está principalmente encaminada a:

- Incorporar nueva funcionalidad al producto software.
- Mejorar la funcionalidad ya presente en el producto software.
- Mejorar el rendimiento del producto software.
- Corregir errores no detectados durante la fase de pruebas del software.

Un sistema software desarrollado atendiendo a los principios de reusabilidad y extensibilidad, garantizará un proceso de mantenimiento flexible. Comúnmente, el mantenimiento del software está orientado a la incorporación de mejoras a la funcionalidad del sistema software, y no tanto a la corrección de errores. La Tabla 12.1 describe los tipos de mantenimiento que comúnmente toman lugar sobre un sistema de software [7,8].

**Tabla 12.1** Tipos de Mantenimiento del Software.

Tipo de mantenimiento	Descripción
<b>Mantenimiento para reparar defectos del software</b>	Los defectos del software pueden ser atribuidos a: 1) errores en la codificación, 2) errores en el diseño de los componentes, y 3) errores en la recolección y análisis de los requerimientos. Los errores menos costosos son los errores en la codificación, mientras que los errores más costosos, en cuanto al mantenimiento del sistema, son los errores cometidos durante la recolección y análisis de los requerimientos, ya que éstos podrían exigir un rediseño del sistema de software.
<b>Mantenimiento para adaptar el software a diferentes entornos operativos</b>	Es el mantenimiento que toma lugar cuando cambia el hardware sobre el cual se ejecutaba el software, el sistema operativo o algún otro tipo de software de soporte.
<b>Mantenimiento para añadir nuevas funcionalidades o modificar las funcionalidades existentes del software</b>	Es el mantenimiento que se produce cuando es necesario modificar o incrementar la funcionalidad del sistema, a petición del cliente. Este tipo de mantenimiento comúnmente conlleva a un rediseño del sistema de software, lo cual resulta costoso.

## XII.2 Gestión del mantenimiento

Una tarea primordial que debe preceder a las actividades propias de la gestión del mantenimiento es la predicción del mantenimiento. La predicción del mantenimiento conlleva a anticipar cuestiones claves [8], tales como:

- ¿Qué partes del sistema son más probables de afectarse por las peticiones de cambio?
- ¿Cuántas peticiones de cambio se esperan?
- ¿Qué partes del sistema serán más costosas de mantener?
- ¿Cuáles serán los costos de mantenimiento durante el período de vida del sistema?
- ¿Cuáles serán los costos de mantenimiento del sistema en el próximo año?

El proceso de mantenimiento del software involucra diferentes actividades, tanto a nivel de la organización propietaria o usuaria del sistema de software, como a nivel

del equipo de desarrollo que se encargará del mantenimiento del sistema de software. En [7] se identifican las siguientes actividades, las cuales se describen más a detalle en la Tabla 12.2:

- Organización del mantenimiento.
- Informe o petición del mantenimiento.
- Flujo de sucesos resultante del informe o petición del mantenimiento.
- Registro de información del mantenimiento.
- Evaluación de las actividades de mantenimiento de software.

**Tabla 12.2** Actividades involucradas en el proceso de mantenimiento del software.

<b>Actividades del mantenimiento de software</b>	<b>Descripción</b>
<b>Organización del mantenimiento</b>	Se refiere a la delegación de responsabilidades entre los miembros del equipo de mantenimiento. La organización del mantenimiento evita confusiones y errores y mejora el flujo de actividades. Entre los roles del equipo de mantenimiento se pueden identificar los siguientes: gestor de configuración, controlador de mantenimiento, supervisor del sistema, resto del grupo de desarrollo (analistas, diseñadores, programadores, etc.).
<b>Informe o petición del mantenimiento</b>	Se refiere a una forma estandarizada de presentación de las peticiones de mantenimiento. Dicho informe o petición de mantenimiento garantiza focalizar y comprender el tipo de mantenimiento requerido (mantenimiento para corregir errores, mantenimiento para modificar funcionalidad existente o añadir nueva funcionalidad, etc.).

<b>Flujo de sucesos resultante del informe o petición del mantenimiento</b>	Es la secuencia de actividades a ejecutar como resultado de una petición de mantenimiento. El primer paso debe ser determinar el tipo de mantenimiento que se desea llevar a cabo. Una vez identificado el tipo de mantenimiento, éste debe ser evaluado, para determinar si es factible proceder o no con el mismo. Si se decide proceder con el mantenimiento, entonces se deben priorizar y ejecutar las tareas involucradas en el tipo de mantenimiento.
<b>Registro de información del mantenimiento</b>	Al igual que en el proceso de desarrollo de software, en la gestión del mantenimiento es necesario documentar todas las fases y actividades involucradas en dicho proceso. Tal documentación evita confusiones y errores y mejora el flujo de actividades.
<b>Evaluación de las actividades de mantenimiento de software</b>	Se refiere a la aplicación de métricas para valorar la calidad del proceso de mantenimiento ejecutado. Entre las medidas que pueden ser consideradas, se encuentran las siguientes: <ul style="list-style-type: none"> <li>➤ Número de personas/hora por cada categoría del mantenimiento.</li> <li>➤ Número medio de cambios por programa, funcionalidad, componente, etc.</li> <li>➤ Número medio de personas/hora por funcionalidad añadida o modificada.</li> <li>➤ ...</li> </ul>

### XII.3 Gestión de los cambios y de los reportes de error durante el mantenimiento

Es natural que durante la operación del software se requieran cambios en las funcionalidades del sistema. El proceso de gestión de los cambios inicia cuando se genera una “Solicitud de Cambio”. Cada Solicitud de Cambio debe pasar por un proceso de revisión, y una vez que se analiza la solicitud puede ser *aceptada* o *rechazada*. En caso de que ésta sea aceptada se procede a atenderla en una nueva variante del producto *que no necesariamente substituye a la anterior*.

Por otra parte, cuando se detecta alguna falla del software se genera un “Reporte de Error” con la descripción del problema. Un reporte de error debe contener: información de control, resumen y descripción. La *información de control* contiene un identificador y el nombre del autor del reporte. En el *resumen* se describe el incidente y se enlistan los módulos involucrados. En la *descripción* se especifican

los detalles del incidente, por ejemplo: las entradas, salidas esperadas, salidas obtenidas, anomalías, entorno en el que ocurrió, fecha y hora.

Se debe elaborar un Reporte de Error por cada defecto encontrado y, cada Reporte de Error debe pasar por el proceso de revisión. Al finalizar este proceso, el Reporte de Error tiene un estatus final de "*Aceptado*" o "*Rechazado*".

Para atender los reportes de error, se genera una nueva versión del producto *que substituye a la anterior*. Al principio de la operación del sistema, que es cuando se produce un mayor número de fallas (curva de Rayleigh) se pueden generar varios Reportes de Error antes de generar una nueva versión y, después de generar varias versiones de esta manera, se generarán nuevas versiones cuando se detecte sólo una o unas cuantas fallas aisladas.

Cuando la liberación de una nueva versión implica *cambios mayores* se suele asignar un número entero consecutivo a la nueva versión que se genera. Por ejemplo, si la versión que está en operación es la versión 2, entonces la nueva versión será la 3. Por otro lado, si los cambios son *menores*, es decir, se corrigieron algunos defectos, pero no hay mejoras o modificaciones en el funcionamiento entonces la versión actual se suele incrementar en una décima, por ejemplo, se pasa de la versión 2 a la 2.1. Finalmente, se dice que hay un *cambio micro* cuando solo se corrigieron uno o pocos defectos que involucraron muy pocos cambios. En este caso la versión se incrementaría en una centésima, por ejemplo, se pasa de la versión 2.1 a la 2.1.1. En la práctica, cada compañía tiene sus estándares para nombrar las versiones de sus productos de software y la decisión de nombrar una nueva versión con cambios *mayores*, *menores* o *micro* depende del responsable de cada proyecto en particular.

## XII.4 Referencias del capítulo

1. Cusumano, M.A. The Factory Approach to Large-Scale Software Development: Implications for Strategy, Technology, and Structure. Classic Reprints Series, 2017.
2. Garland, J., Anthony, R. Large Scale Software Architecture. A Practical Guide Using UML. Wiley, 2003.
3. Janakiram, D. Building Large Scale Software Systems. McGraw Hill Education, 2013.
4. Lakos, J. Large-Scale C++ Software Design. Addison Wesley, 1996.
5. Screerer, A. Coordination in Large-Scale Agile Software Development: Integrating Conditions and Configurations in Multiteam Systems (Progress in IS). Springer, 2017.
6. Pfleeger, S. L. Ingeniería de software: Teoría y práctica. Pearson Education, 2002.
7. Pressman, R. S. Ingeniería del software: Un enfoque práctico. McGraw-Hill, 2010.
8. Sommerville, I. Ingeniería del software. Pearson Addison Wesley, 2012.
9. Tsui, F., Karam, O., Bernal, B. Essentials of software engineering. Jonas & Bartlett Learning books, 2015.
10. Van Vliet, H. Software engineering: Principles and practice. John Wiley & Sons, 2000.

# Apéndice

## A. Glosario de Abreviaciones

Abreviación	Descripción
<b>AC</b>	Acústica – Componente de detección de fugas en ductos que transportan hidrocarburos, basado en la tecnología Acústica. Se hace referencia a esta sigla en uno de los dos casos de estudio tratados al final de la gran mayoría de los capítulos.
<b>AOO</b>	Análisis Orientado a Objetos.
<b>BD</b>	Base de datos.
<b>COCOMO</b>	Modelo Constructivo de Costo. Del inglés, <i>Constructive Cost Model</i> .
<b>DOO</b>	Diseño Orientado a Objetos.
<b>DSD</b>	Diagramas de Secuencia Detallados.
<b>DTIU</b>	Diagramas de Transiciones entre Interfaces de Usuario.
<b>FO</b>	Fibra Óptica – Componente de detección de fugas en ductos que transportan hidrocarburos, basado en la tecnología Fibra Óptica. Se hace referencia a esta sigla en uno de los dos casos de estudio tratados al final de la gran mayoría de los capítulos.
<b>GUI</b>	Interfaz Gráfica de Usuario, del inglés <i>Graphical User Interface</i> .
<b>HMVC</b>	Patrón arquitectónico Modelo-Vista-Controlador Jerárquico, del inglés <i>Hierarchical Model-View-Control</i> .
<b>IDE</b>	Ambiente Integrado de Desarrollo, del inglés <i>Integrated Development Environment</i> .
<b>LC</b>	Lazo Cerrado – Componente de detección de fugas en ductos que transportan hidrocarburos, basado en la técnica Balance de Masa. Se hace referencia a esta sigla en uno de los dos casos de estudio tratados al final de la gran mayoría de los capítulos.
<b>MVC</b>	Patrón arquitectónico Modelo-Vista-Controlador, del inglés <i>Model-View-Controller</i> .

<b>MVP</b>	Patrón arquitectónico Modelo-Vista-Presentación, del inglés <i>Model-View-Presenter</i> .
<b>OO</b>	Orientado a Objetos.
<b>PAC</b>	Modelo arquitectónico Presentación-Abstracción-Control.
<b>POO</b>	Programación Orientada a Objetos.
<b>RPS-GS</b>	Recuperación de Proyectos de Software. La sigla hace referencia al nombre del sistema de software para tal finalidad.
<b>SO</b>	Sistema Operativo.
<b>SPI</b>	Subsistema de Procesamiento Integral de Datos del Sistema de Detección de Fugas en Ductos que Transportan Hidrocarburos. Corresponde a uno de los dos casos de estudio utilizados para ilustrar los contenidos de la gran mayoría de los capítulos.
<b>VV</b>	Vigilantes Virtuales – Componente de detección de fugas en ductos que transportan hidrocarburos, basado en la técnica Balance de Masa. Se hace referencia a esta sigla en uno de los dos casos de estudio tratados al final de la gran mayoría de los capítulos.
<b>UP</b>	Proceso Unificado de Desarrollo de Software, del inglés <i>Unified Process</i> .

## B. Glosario de Términos

<b>Término</b>	<b>Descripción</b>
<b>Análisis inteligente de información</b>	Técnica orientada a la extracción de información y conocimiento de grandes volúmenes de datos, con la finalidad de utilizar estos resultados en la toma de decisiones. A partir del uso de técnicas, comúnmente basadas en el aprendizaje automatizado, los datos se transforman en conocimiento.
<b>Minería de datos</b>	La minería de datos significa análisis inteligente de la información. La minería de datos utiliza técnicas basadas en inteligencia artificial, aprendizaje automatizado y estadística, para extraer conocimiento de grandes volúmenes de datos. Una etapa crucial de la minería de datos es la preparación de los datos, la cual incluye la limpieza y completamiento de los

	datos, como paso previo a que éstos sean u para la construcción del modelo de predicción, diagnóstico, clasificación, etc.
<b>Red Neuronal Artificial</b>	Las redes neuronales artificiales son sistemas de procesamiento de información, cuya esencia central es la analogía que éstas exhiben con la estructura y funcionamiento de las redes neuronales biológicas. Estos sistemas de procesamiento de información son concebidos para emular una serie de funciones llevadas a cabo por el cerebro, tales como: reconocimiento de patrones, memoria, aprendizaje y la generalización o abstracción del conocimiento, entre otras. Una red neuronal artificial consiste de un conjunto de unidades de procesamiento (conocidas como neuronas artificiales), las cuales se conectan entre sí siguiendo diferentes patrones de interconexión. Comúnmente, las neuronas artificiales se organizan en capas o estratos.
<b>Sistemas expertos</b>	Un sistema experto es un sistema informático con conocimientos en la solución de problemas. Esto es, un sistema que posee conocimientos acerca de un dominio particular, que puede comprender problemas de dicho dominio, y que posee métodos de inferencia o razonamiento para manipular este conocimiento y resolver los problemas de la misma forma en que lo haría un experto humano. Los sistemas expertos constituyen una de las principales áreas en las que se focalizó la inteligencia artificial en sus inicios.



Casa abierta al tiempo

**UNIVERSIDAD AUTÓNOMA METROPOLITANA**  
**Unidad Cuajimalpa**

